

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

Documentation Changes

May 2020

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: 252046-064



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 © 1997-2020, 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	<ul style="list-style-type: none">Initial release	November 2002
-002	<ul style="list-style-type: none">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	<ul style="list-style-type: none">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	<ul style="list-style-type: none">Removed Documentation changes 1-17.Added Documentation changes 1-24.	June 2003
-005	<ul style="list-style-type: none">Removed Documentation Changes 1-24.Added Documentation Changes 1-15.	September 2003
-006	<ul style="list-style-type: none">Added Documentation Changes 16- 34.	November 2003
-007	<ul style="list-style-type: none">Updated Documentation changes 14, 16, 17, and 28.Added Documentation Changes 35-45.	January 2004
-008	<ul style="list-style-type: none">Removed Documentation Changes 1-45.Added Documentation Changes 1-5.	March 2004
-009	<ul style="list-style-type: none">Added Documentation Changes 7-27.	May 2004
-010	<ul style="list-style-type: none">Removed Documentation Changes 1-27.Added Documentation Changes 1.	August 2004
-011	<ul style="list-style-type: none">Added Documentation Changes 2-28.	November 2004
-012	<ul style="list-style-type: none">Removed Documentation Changes 1-28.Added Documentation Changes 1-16.	March 2005
-013	<ul style="list-style-type: none">Updated title.There are no Documentation Changes for this revision of the document.	July 2005
-014	<ul style="list-style-type: none">Added Documentation Changes 1-21.	September 2005
-015	<ul style="list-style-type: none">Removed Documentation Changes 1-21.Added Documentation Changes 1-20.	March 9, 2006
-016	<ul style="list-style-type: none">Added Documentation changes 21-23.	March 27, 2006
-017	<ul style="list-style-type: none">Removed Documentation Changes 1-23.Added Documentation Changes 1-36.	September 2006
-018	<ul style="list-style-type: none">Added Documentation Changes 37-42.	October 2006
-019	<ul style="list-style-type: none">Removed Documentation Changes 1-42.Added Documentation Changes 1-19.	March 2007
-020	<ul style="list-style-type: none">Added Documentation Changes 20-27.	May 2007
-021	<ul style="list-style-type: none">Removed Documentation Changes 1-27.Added Documentation Changes 1-6	November 2007
-022	<ul style="list-style-type: none">Removed Documentation Changes 1-6Added Documentation Changes 1-6	August 2008
-023	<ul style="list-style-type: none">Removed Documentation Changes 1-6Added Documentation Changes 1-21	March 2009



Revision	Description	Date
-024	<ul style="list-style-type: none"> Removed Documentation Changes 1-21 Added Documentation Changes 1-16 	June 2009
-025	<ul style="list-style-type: none"> Removed Documentation Changes 1-16 Added Documentation Changes 1-18 	September 2009
-026	<ul style="list-style-type: none"> Removed Documentation Changes 1-18 Added Documentation Changes 1-15 	December 2009
-027	<ul style="list-style-type: none"> Removed Documentation Changes 1-15 Added Documentation Changes 1-24 	March 2010
-028	<ul style="list-style-type: none"> Removed Documentation Changes 1-24 Added Documentation Changes 1-29 	June 2010
-029	<ul style="list-style-type: none"> Removed Documentation Changes 1-29 Added Documentation Changes 1-29 	September 2010
-030	<ul style="list-style-type: none"> Removed Documentation Changes 1-29 Added Documentation Changes 1-29 	January 2011
-031	<ul style="list-style-type: none"> Removed Documentation Changes 1-29 Added Documentation Changes 1-29 	April 2011
-032	<ul style="list-style-type: none"> Removed Documentation Changes 1-29 Added Documentation Changes 1-14 	May 2011
-033	<ul style="list-style-type: none"> Removed Documentation Changes 1-14 Added Documentation Changes 1-38 	October 2011
-034	<ul style="list-style-type: none"> Removed Documentation Changes 1-38 Added Documentation Changes 1-16 	December 2011
-035	<ul style="list-style-type: none"> Removed Documentation Changes 1-16 Added Documentation Changes 1-18 	March 2012
-036	<ul style="list-style-type: none"> Removed Documentation Changes 1-18 Added Documentation Changes 1-17 	May 2012
-037	<ul style="list-style-type: none"> Removed Documentation Changes 1-17 Added Documentation Changes 1-28 	August 2012
-038	<ul style="list-style-type: none"> Removed Documentation Changes 1-28 Add Documentation Changes 1-22 	January 2013
-039	<ul style="list-style-type: none"> Removed Documentation Changes 1-22 Add Documentation Changes 1-17 	June 2013
-040	<ul style="list-style-type: none"> Removed Documentation Changes 1-17 Add Documentation Changes 1-24 	September 2013
-041	<ul style="list-style-type: none"> Removed Documentation Changes 1-24 Add Documentation Changes 1-20 	February 2014
-042	<ul style="list-style-type: none"> Removed Documentation Changes 1-20 Add Documentation Changes 1-8 	February 2014
-043	<ul style="list-style-type: none"> Removed Documentation Changes 1-8 Add Documentation Changes 1-43 	June 2014
-044	<ul style="list-style-type: none"> Removed Documentation Changes 1-43 Add Documentation Changes 1-12 	September 2014
-045	<ul style="list-style-type: none"> Removed Documentation Changes 1-12 Add Documentation Changes 1-22 	January 2015
-046	<ul style="list-style-type: none"> Removed Documentation Changes 1-22 Add Documentation Changes 1-25 	April 2015
-047	<ul style="list-style-type: none"> Removed Documentation Changes 1-25 Add Documentation Changes 1-19 	June 2015



Revision	Description	Date
-048	<ul style="list-style-type: none">Removed Documentation Changes 1-19Add Documentation Changes 1-33	September 2015
-049	<ul style="list-style-type: none">Removed Documentation Changes 1-33Add Documentation Changes 1-33	December 2015
-050	<ul style="list-style-type: none">Removed Documentation Changes 1-33Add Documentation Changes 1-9	April 2016
-051	<ul style="list-style-type: none">Removed Documentation Changes 1-9Add Documentation Changes 1-20	June 2016
-052	<ul style="list-style-type: none">Removed Documentation Changes 1-20Add Documentation Changes 1-22	September 2016
-053	<ul style="list-style-type: none">Removed Documentation Changes 1-22Add Documentation Changes 1-26	December 2016
-054	<ul style="list-style-type: none">Removed Documentation Changes 1-26Add Documentation Changes 1-20	March 2017
-055	<ul style="list-style-type: none">Removed Documentation Changes 1-20Add Documentation Changes 1-28	July 2017
-056	<ul style="list-style-type: none">Removed Documentation Changes 1-28Add Documentation Changes 1-18	October 2017
-057	<ul style="list-style-type: none">Removed Documentation Changes 1-18Add Documentation Changes 1-29	December 2017
-058	<ul style="list-style-type: none">Removed Documentation Changes 1-29Add Documentation Changes 1-17	March 2018
-059	<ul style="list-style-type: none">Removed Documentation Changes 1-17Add Documentation Changes 1-24	May 2018
-060	<ul style="list-style-type: none">Removed Documentation Changes 1-24Add Documentation Changes 1-23	November 2018
-061	<ul style="list-style-type: none">Removed Documentation Changes 1-23Add Documentation Changes 1-21	January 2019
-062	<ul style="list-style-type: none">Removed Documentation Changes 1-21Add Documentation Changes 1-28	May 2019
-063	<ul style="list-style-type: none">Removed Documentation Changes 1-28Add Documentation Changes 1-34	October 2019
-064	<ul style="list-style-type: none">Removed Documentation Changes 1-34Add Documentation Changes 1-36	May 2020

§

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

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

Documentation Changes(Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
27	Updates to Chapter 25, Volume 3C
28	Updates to Chapter 26, Volume 3C
29	Updates to Chapter 27, Volume 3C
30	Updates to Chapter 28, Volume 3C
31	Updates to Chapter 29, Volume 3C
32	Updates to Chapter 35, Volume 3C
33	Updates to Chapter 37, Volume 3D
34	Updates to Chapter 40, Volume 3D
35	Updates to Appendix B, Volume 3D
36	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 2, Volume 1

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

Changes to this chapter: Update to Table 2-4 "Proposed Intel ISA and Features Removal List" and update to Table 2-5 "Intel ISA and Features Removal List".

2.1 BRIEF HISTORY OF INTEL® 64 AND IA-32 ARCHITECTURE

The following sections provide a summary of the major technical evolutions from IA-32 to Intel 64 architecture: starting from the Intel 8086 processor to the latest Intel® Core® 2 Duo, Core 2 Quad and Intel Xeon processor 5300 and 7300 series. Object code created for processors released as early as 1978 still executes on the latest processors in the Intel 64 and IA-32 architecture families.

2.1.1 16-bit Processors and Segmentation (1978)

The IA-32 architecture family was preceded by 16-bit processors, the 8086 and 8088. The 8086 has 16-bit registers and a 16-bit external data bus, with 20-bit addressing giving a 1-MByte address space. The 8088 is similar to the 8086 except it has an 8-bit external data bus.

The 8086/8088 introduced segmentation to the IA-32 architecture. With segmentation, a 16-bit segment register contains a pointer to a memory segment of up to 64 KBytes. Using four segment registers at a time, 8086/8088 processors are able to address up to 256 KBytes without switching between segments. The 20-bit addresses that can be formed using a segment register and an additional 16-bit pointer provide a total address range of 1 MByte.

2.1.2 The Intel® 286 Processor (1982)

The Intel 286 processor introduced protected mode operation into the IA-32 architecture. Protected mode uses the segment register content as selectors or pointers into descriptor tables. Descriptors provide 24-bit base addresses with a physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and a number of protection mechanisms. These mechanisms include:

- Segment limit checking
- Read-only and execute-only segment options
- Four privilege levels

2.1.3 The Intel386™ Processor (1985)

The Intel386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers for use both to hold operands and for addressing. The lower half of each 32-bit Intel386 register retains the properties of the 16-bit registers of earlier generations, permitting backward compatibility. The processor also provides a virtual-8086 mode that allows for even greater efficiency when executing programs created for 8086/8088 processors.

In addition, the Intel386 processor has support for:

- A 32-bit address bus that supports up to 4-GBytes of physical memory
- A segmented-memory model and a flat memory model
- Paging, with a fixed 4-KByte page size providing a method for virtual memory management
- Support for parallel stages

2.1.4 The Intel486™ Processor (1989)

The Intel486™ processor added more parallel execution capability by expanding the Intel386 processor's instruction decode and execution units into five pipelined stages. Each stage operates in parallel with the others on up to five instructions in different stages of execution.

In addition, the processor added:

- An 8-KByte on-chip first-level cache that increased the percent of instructions that could execute at the scalar rate of one per clock
- An integrated x87 FPU
- Power saving and system management capabilities

2.1.5 The Intel® Pentium® Processor (1993)

The introduction of the Intel Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, together can execute two instructions per clock). The on-chip first-level cache doubled, with 8 KBytes devoted to code and another 8 KBytes devoted to data. The data cache uses the MESI protocol to support more efficient write-back cache in addition to the write-through cache previously used by the Intel486 processor. Branch prediction with an on-chip branch table was added to increase performance in looping constructs.

In addition, the processor added:

- Extensions to make the virtual-8086 mode more efficient and allow for 4-MByte as well as 4-KByte pages
- Internal data paths of 128 and 256 bits add speed to internal data transfers
- Burstable external data bus was increased to 64 bits
- An APIC to support systems with multiple processors
- A dual processor mode to support glueless two processor systems

A subsequent stepping of the Pentium family introduced Intel MMX technology (the Pentium Processor with MMX technology). Intel MMX technology uses the single-instruction, multiple-data (SIMD) execution model to perform parallel computations on packed integer data contained in 64-bit registers.

See Section 2.2.7, "SIMD Instructions."

2.1.6 The P6 Family of Processors (1995-1999)

The P6 family of processors was based on a superscalar microarchitecture that set new performance standards; see also Section 2.2.1, "P6 Family Microarchitecture." One of the goals in the design of the P6 family microarchitecture was to exceed the performance of the Pentium processor significantly while using the same 0.6-micrometer, four-layer, metal BICMOS manufacturing process. Members of this family include the following:

- The **Intel Pentium Pro processor** is three-way superscalar. Using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. The Pentium Pro introduced the dynamic execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a superscalar implementation. The processor was further enhanced by its caches. It has the same two on-chip 8-KByte 1st-Level caches as the Pentium processor and an additional 256-KByte Level 2 cache in the same package as the processor.
- The **Intel Pentium II processor** added Intel MMX technology to the P6 family processors along with new packaging and several hardware enhancements. The processor core is packaged in the single edge contact cartridge (SECC). The Level 1 data and instruction caches were enlarged to 16 KBytes each, and Level 2 cache sizes of 256 KBytes, 512 KBytes, and 1 MByte are supported. A half-frequency backside bus connects the Level 2 cache to the processor. Multiple low-power states such as AutoHALT, Stop-Grant, Sleep, and Deep Sleep are supported to conserve power when idling.
- The **Pentium II Xeon processor** combined the premium characteristics of previous generations of Intel processors. This includes: 4-way, 8-way (and up) scalability and a 2 MByte 2nd-Level cache running on a full-frequency backside bus.
- The **Intel Celeron processor** family focused on the value PC market segment. Its introduction offers an integrated 128 KBytes of Level 2 cache and a plastic pin grid array (P.P.G.A.) form factor to lower system design cost.
- The **Intel Pentium III processor** introduced the Streaming SIMD Extensions (SSE) to the IA-32 architecture. SSE extensions expand the SIMD execution model introduced with the Intel MMX technology by providing a

new set of 128-bit registers and the ability to perform SIMD operations on packed single-precision floating-point values. See Section 2.2.7, “SIMD Instructions.”

- The **Pentium III Xeon processor** extended the performance levels of the IA-32 processors with the enhancement of a full-speed, on-die, and Advanced Transfer Cache.

2.1.7 The Intel® Pentium® 4 Processor Family (2000-2006)

The Intel Pentium 4 processor family is based on Intel NetBurst microarchitecture; see Section 2.2.2, “Intel NetBurst® Microarchitecture.”

The Intel Pentium 4 processor introduced Streaming SIMD Extensions 2 (SSE2); see Section 2.2.7, “SIMD Instructions.” The Intel Pentium 4 processor 3.40 GHz, supporting Hyper-Threading Technology introduced Streaming SIMD Extensions 3 (SSE3); see Section 2.2.7, “SIMD Instructions.”

Intel 64 architecture was introduced in the Intel Pentium 4 Processor Extreme Edition supporting Hyper-Threading Technology and in the Intel Pentium 4 Processor 6xx and 5xx sequences.

Intel® Virtualization Technology (Intel® VT) was introduced in the Intel Pentium 4 processor 672 and 662.

2.1.8 The Intel® Xeon® Processor (2001- 2007)

Intel Xeon processors (with exception for dual-core Intel Xeon processor LV, Intel Xeon processor 5100 series) are based on the Intel NetBurst microarchitecture; see Section 2.2.2, “Intel NetBurst® Microarchitecture.” As a family, this group of IA-32 processors (more recently Intel 64 processors) is designed for use in multi-processor server systems and high-performance workstations.

The Intel Xeon processor MP introduced support for Intel® Hyper-Threading Technology; see Section 2.2.8, “Intel® Hyper-Threading Technology.”

The 64-bit Intel Xeon processor 3.60 GHz (with an 800 MHz System Bus) was used to introduce Intel 64 architecture. The Dual-Core Intel Xeon processor includes dual core technology. The Intel Xeon processor 70xx series includes Intel Virtualization Technology.

The Intel Xeon processor 5100 series introduces power-efficient, high performance Intel Core microarchitecture. This processor is based on Intel 64 architecture; it includes Intel Virtualization Technology and dual-core technology. The Intel Xeon processor 3000 series are also based on Intel Core microarchitecture. The Intel Xeon processor 5300 series introduces four processor cores in a physical package, they are also based on Intel Core microarchitecture.

2.1.9 The Intel® Pentium® M Processor (2003-2006)

The Intel Pentium M processor family is a high performance, low power mobile processor family with microarchitectural enhancements over previous generations of IA-32 Intel mobile processors. This family is designed for extending battery life and seamless integration with platform innovations that enable new usage models (such as extended mobility, ultra thin form-factors, and integrated wireless networking).

Its enhanced microarchitecture includes:

- Support for Intel Architecture with Dynamic Execution
- A high performance, low-power core manufactured using Intel’s advanced process technology with copper interconnect
- On-die, primary 32-KByte instruction cache and 32-KByte write-back data cache
- On-die, second-level cache (up to 2 MByte) with Advanced Transfer Cache Architecture
- Advanced Branch Prediction and Data Prefetch Logic
- Support for MMX technology, Streaming SIMD instructions, and the SSE2 instruction set
- A 400 or 533 MHz, Source-Synchronous Processor System Bus
- Advanced power management using Enhanced Intel SpeedStep® technology

2.1.10 The Intel® Pentium® Processor Extreme Edition (2005)

The Intel Pentium processor Extreme Edition introduced dual-core technology. This technology provides advanced hardware multi-threading support. The processor is based on Intel NetBurst microarchitecture and supports SSE, SSE2, SSE3, Hyper-Threading Technology, and Intel 64 architecture.

See also:

- Section 2.2.2, “Intel NetBurst® Microarchitecture”
- Section 2.2.3, “Intel® Core™ Microarchitecture”
- Section 2.2.7, “SIMD Instructions”
- Section 2.2.8, “Intel® Hyper-Threading Technology”
- Section 2.2.9, “Multi-Core Technology”
- Section 2.2.10, “Intel® 64 Architecture”

2.1.11 The Intel® Core™ Duo and Intel® Core™ Solo Processors (2006-2007)

The Intel Core Duo processor offers power-efficient, dual-core performance with a low-power design that extends battery life. This family and the single-core Intel Core Solo processor offer microarchitectural enhancements over Pentium M processor family.

Its enhanced microarchitecture includes:

- Intel® Smart Cache which allows for efficient data sharing between two processor cores
- Improved decoding and SIMD execution
- Intel® Dynamic Power Coordination and Enhanced Intel® Deeper Sleep to reduce power consumption
- Intel® Advanced Thermal Manager which features digital thermal sensor interfaces
- Support for power-optimized 667 MHz bus

The dual-core Intel Xeon processor LV is based on the same microarchitecture as Intel Core Duo processor, and supports IA-32 architecture.

2.1.12 The Intel® Xeon® Processor 5100, 5300 Series and Intel® Core™ 2 Processor Family (2006)

The Intel Xeon processor 3000, 3200, 5100, 5300, and 7300 series, Intel Pentium Dual-Core, Intel Core 2 Extreme, Intel Core 2 Quad processors, and Intel Core 2 Duo processor family support Intel 64 architecture; they are based on the high-performance, power-efficient Intel® Core microarchitecture built on 65 nm process technology. The Intel Core microarchitecture includes the following innovative features:

- Intel® Wide Dynamic Execution to increase performance and execution throughput
- Intel® Intelligent Power Capability to reduce power consumption
- Intel® Advanced Smart Cache which allows for efficient data sharing between two processor cores
- Intel® Smart Memory Access to increase data bandwidth and hide latency of memory accesses
- Intel® Advanced Digital Media Boost which improves application performance using multiple generations of Streaming SIMD extensions

The Intel Xeon processor 5300 series, Intel Core 2 Extreme processor QX6800 series, and Intel Core 2 Quad processors support Intel quad-core technology.

2.1.13 The Intel® Xeon® Processor 5200, 5400, 7400 Series and Intel® Core™ 2 Processor Family (2007)

The Intel Xeon processor 5200, 5400, and 7400 series, Intel Core 2 Quad processor Q9000 Series, Intel Core 2 Duo processor E8000 series support Intel 64 architecture; they are based on the Enhanced Intel® Core microarchitec-

ture using 45 nm process technology. The Enhanced Intel Core microarchitecture provides the following improved features:

- A radix-16 divider, faster OS primitives further increases the performance of Intel® Wide Dynamic Execution.
- Improves Intel® Advanced Smart Cache with Up to 50% larger level-two cache and up to 50% increase in way-set associativity.
- A 128-bit shuffler engine significantly improves the performance of Intel® Advanced Digital Media Boost and SSE4.

Intel Xeon processor 5400 series and Intel Core 2 Quad processor Q9000 Series support Intel quad-core technology. Intel Xeon processor 7400 series offers up to six processor cores and an L3 cache up to 16 MBytes.

2.1.14 The Intel® Atom™ Processor Family (2008)

The first generation of Intel® Atom™ processors are built on 45 nm process technology. They are based on a new microarchitecture, Intel® Atom™ microarchitecture, which is optimized for ultra low power devices. The Intel® Atom™ microarchitecture features two in-order execution pipelines that minimize power consumption, increase battery life, and enable ultra-small form factors. The initial Intel Atom Processor family and subsequent generations including Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series provide the following features:

- Enhanced Intel® SpeedStep® Technology
- Intel® Hyper-Threading Technology
- Deep Power Down Technology with Dynamic Cache Sizing
- Support for instruction set extensions up to and including Supplemental Streaming SIMD Extensions 3 (SSSE3).
- Support for Intel® Virtualization Technology
- Support for Intel® 64 Architecture (excluding Intel Atom processor Z5xx Series)

2.1.15 The Intel® Atom™ Processor Family Based on Silvermont Microarchitecture (2013)

Intel Atom Processor C2xxx, E3xxx, S1xxx series are based on the Silvermont microarchitecture. Processors based on the Silvermont microarchitecture supports instruction set extensions up to and including SSE4.2, AESNI, and PCLMULQDQ.

2.1.16 The Intel® Core™ i7 Processor Family (2008)

The Intel Core i7 processor 900 series support Intel 64 architecture; they are based on Intel® microarchitecture code name Nehalem using 45 nm process technology. The Intel Core i7 processor and Intel Xeon processor 5500 series include the following innovative features:

- Intel® Turbo Boost Technology converts thermal headroom into higher performance.
- Intel® HyperThreading Technology in conjunction with Quadcore to provide four cores and eight threads.
- Dedicated power control unit to reduce active and idle power consumption.
- Integrated memory controller on the processor supporting three channel of DDR3 memory.
- 8 MB inclusive Intel® Smart Cache.
- Intel® QuickPath interconnect (QPI) providing point-to-point link to chipset.
- Support for SSE4.2 and SSE4.1 instruction sets.
- Second generation Intel Virtualization Technology.

2.1.17 The Intel® Xeon® Processor 7500 Series (2010)

The Intel Xeon processor 7500 and 6500 series are based on Intel microarchitecture code name Nehalem using 45 nm process technology. They support the same features described in Section 2.1.16, plus the following innovative features:

- Up to eight cores per physical processor package.
- Up to 24 MB inclusive Intel® Smart Cache.
- Provides Intel® Scalable Memory Interconnect (Intel® SMI) channels with Intel® 7500 Scalable Memory Buffer to connect to system memory.
- Advanced RAS supporting software recoverable machine check architecture.

2.1.18 2010 Intel® Core™ Processor Family (2010)

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

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

2.1.19 The Intel® Xeon® Processor 5600 Series (2010)

The Intel Xeon processor 5600 series are based on Intel microarchitecture code name Westmere using 32 nm process technology. They support the same features described in Section 2.1.16, plus the following innovative features:

- Up to six cores per physical processor package.
- Up to 12 MB enhanced Intel® Smart Cache.
- Support for AESNI, PCLMULQDQ, SSE4.2 and SSE4.1 instruction sets.
- Flexible Intel Virtualization Technologies across processor and I/O.

2.1.20 The Second Generation Intel® Core™ Processor Family (2011)

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

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

Intel Xeon processor E3-1200 product family is also based on the Sandy Bridge microarchitecture.

Intel Xeon processor E5-2400/1400 product families are based on the Sandy Bridge-EP microarchitecture.

Intel Xeon processor E5-4600/2600/1600 product families are based on the Sandy Bridge-EP microarchitecture and provide support for multiple sockets.

2.1.21 The Third Generation Intel® Core™ Processor Family (2012)

The Third Generation Intel Core processor family spans Intel Core i7, i5 and i3 processors based on the Ivy Bridge microarchitecture. The Intel Xeon processor E7-8800/4800/2800 v2 product families and Intel Xeon processor E3-1200 v2 product family are also based on the Ivy Bridge microarchitecture.

The Intel Xeon processor E5-2400/1400 v2 product families are based on the Ivy Bridge-EP microarchitecture.

The Intel Xeon processor E5-4600/2600/1600 v2 product families are based on the Ivy Bridge-EP microarchitecture and provide support for multiple sockets.

2.1.22 The Fourth Generation Intel® Core™ Processor Family (2013)

The Fourth Generation Intel Core processor family spans Intel Core i7, i5 and i3 processors based on the Haswell microarchitecture. Intel Xeon processor E3-1200 v3 product family is also based on the Haswell microarchitecture.

2.2 MORE ON SPECIFIC ADVANCES

The following sections provide more information on major innovations.

2.2.1 P6 Family Microarchitecture

The Pentium Pro processor introduced a new microarchitecture commonly referred to as P6 processor microarchitecture. The P6 processor microarchitecture was later enhanced with an on-die, Level 2 cache, called Advanced Transfer Cache.

The microarchitecture is a three-way superscalar, pipelined architecture. Three-way superscalar means that by using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. To handle this level of instruction throughput, the P6 processor family uses a decoupled, 12-stage superpipeline that supports out-of-order instruction execution.

Figure 2-1 shows a conceptual view of the P6 processor microarchitecture pipeline with the Advanced Transfer Cache enhancement.

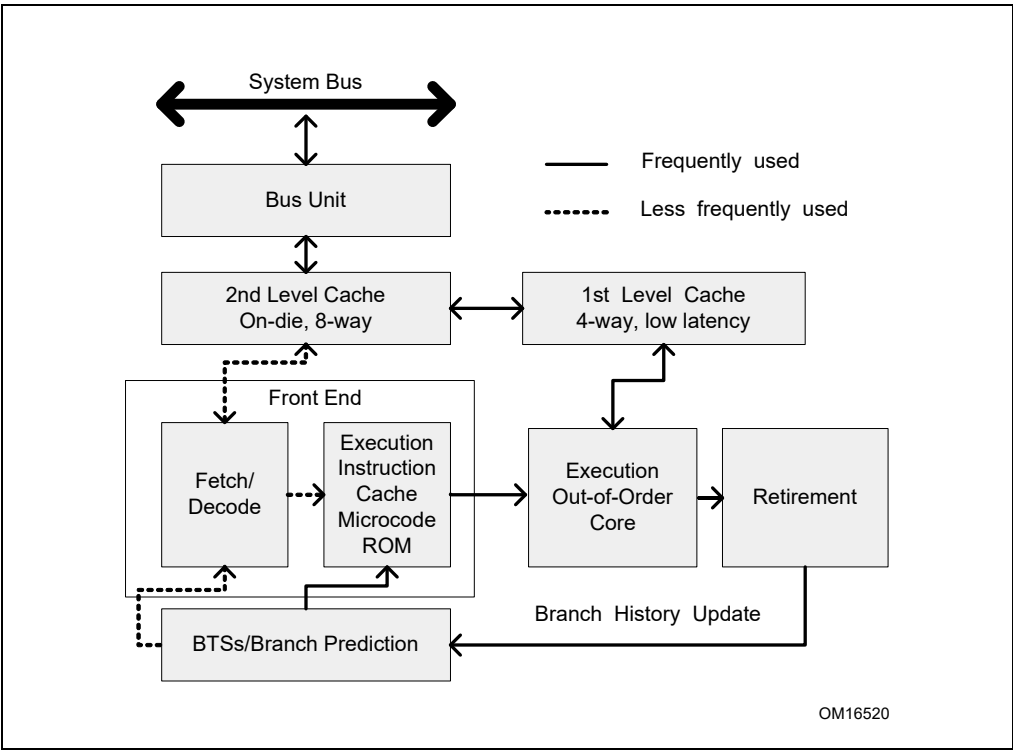


Figure 2-1. The P6 Processor Microarchitecture with Advanced Transfer Cache Enhancement

To ensure a steady supply of instructions and data for the instruction execution pipeline, the P6 processor microarchitecture incorporates two cache levels. The Level 1 cache provides an 8-KByte instruction cache and an 8-KByte

data cache, both closely coupled to the pipeline. The Level 2 cache provides 256-KByte, 512-KByte, or 1-MByte static RAM that is coupled to the core processor through a full clock-speed 64-bit cache bus.

The centerpiece of the P6 processor microarchitecture is an out-of-order execution mechanism called dynamic execution. Dynamic execution incorporates three data-processing concepts:

- **Deep branch prediction** allows the processor to decode instructions beyond branches to keep the instruction pipeline full. The P6 processor family implements highly optimized branch prediction algorithms to predict the direction of the instruction.
- **Dynamic data flow analysis** requires real-time analysis of the flow of data through the processor to determine dependencies and to detect opportunities for out-of-order instruction execution. The out-of-order execution core can monitor many instructions and execute these instructions in the order that best optimizes the use of the processor's multiple execution units, while maintaining the data integrity.
- **Speculative execution** refers to the processor's ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream. To make speculative execution possible, the P6 processor microarchitecture decouples the dispatch and execution of instructions from the commitment of results. The processor's out-of-order execution core uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers. The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions. When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the IA-32 registers (the processor's eight general-purpose registers and eight x87 FPU data registers) in the order they were originally issued and retires the instructions from the instruction pool.

2.2.2 Intel NetBurst® Microarchitecture

The Intel NetBurst microarchitecture provides:

- The Rapid Execution Engine
 - Arithmetic Logic Units (ALUs) run at twice the processor frequency
 - Basic integer operations can dispatch in 1/2 processor clock tick
- Hyper-Pipelined Technology
 - Deep pipeline to enable industry-leading clock rates for desktop PCs and servers
 - Frequency headroom and scalability to continue leadership into the future
- Advanced Dynamic Execution
 - Deep, out-of-order, speculative execution engine
 - Up to 126 instructions in flight
 - Up to 48 loads and 24 stores in pipeline¹
 - Enhanced branch prediction capability
 - Reduces the misprediction penalty associated with deeper pipelines
 - Advanced branch prediction algorithm
 - 4K-entry branch target array
- New cache subsystem
 - First level caches
 - Advanced Execution Trace Cache stores decoded instructions
 - Execution Trace Cache removes decoder latency from main execution loops
 - Execution Trace Cache integrates path of program execution flow into a single line

1. Intel 64 and IA-32 processors based on the Intel NetBurst microarchitecture at 90 nm process can handle more than 24 stores in flight.

- Low latency data cache
- Second level cache
 - Full-speed, unified 8-way Level 2 on-die Advance Transfer Cache
 - Bandwidth and performance increases with processor frequency
- High-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus
 - Supports quad-pumped, scalable bus clock to achieve up to 4X effective speed
 - Capable of delivering up to 8.5 GBytes of bandwidth per second
- Superscalar issue to enable parallelism
- Expanded hardware registers with renaming to avoid register name space limitations
- 64-byte cache line size (transfers data up to two lines per sector)

Figure 2-2 is an overview of the Intel NetBurst microarchitecture. This microarchitecture pipeline is made up of three sections: (1) the front end pipeline, (2) the out-of-order execution core, and (3) the retirement unit.

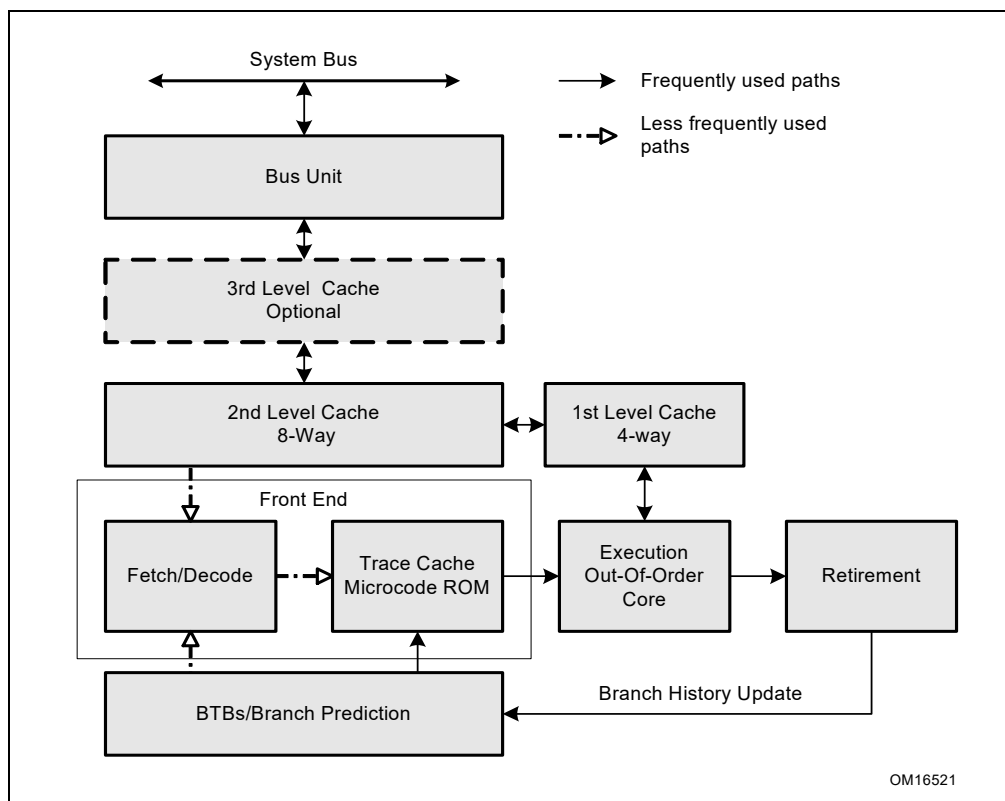


Figure 2-2. The Intel NetBurst Microarchitecture

2.2.2.1 The Front End Pipeline

The front end supplies instructions in program order to the out-of-order execution core. It performs a number of functions:

- Prefetches instructions that are likely to be executed
- Fetches instructions that have not already been prefetched
- Decodes instructions into micro-operations
- Generates microcode for complex instructions and special-purpose code
- Delivers decoded instructions from the execution trace cache

- Predicts branches using highly advanced algorithm

The pipeline is designed to address common problems in high-speed, pipelined microprocessors. Two of these problems contribute to major sources of delays:

- time to decode instructions fetched from the target
- wasted decode bandwidth due to branches or branch target in the middle of cache lines

The operation of the pipeline's trace cache addresses these issues. Instructions are constantly being fetched and decoded by the translation engine (part of the fetch/decode logic) and built into sequences of micro-ops called traces. At any time, multiple traces (representing prefetched branches) are being stored in the trace cache. The trace cache is searched for the instruction that follows the active branch. If the instruction also appears as the first instruction in a pre-fetched branch, the fetch and decode of instructions from the memory hierarchy ceases and the pre-fetched branch becomes the new source of instructions (see Figure 2-2).

The trace cache and the translation engine have cooperating branch prediction hardware. Branch targets are predicted based on their linear addresses using branch target buffers (BTBs) and fetched as soon as possible.

2.2.2.2 Out-Of-Order Execution Core

The out-of-order execution core's ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one micro-op is delayed, other micro-ops may proceed around it. The processor employs several buffers to smooth the flow of micro-ops.

The core is designed to facilitate parallel execution. It can dispatch up to six micro-ops per cycle (this exceeds trace cache and retirement micro-op bandwidth). Most pipelines can start executing a new micro-op every cycle, so several instructions can be in flight at a time for each pipeline. A number of arithmetic logical unit (ALU) instructions can start at two per cycle; many floating-point instructions can start once every two cycles.

2.2.2.3 Retirement Unit

The retirement unit receives the results of the executed micro-ops from the out-of-order execution core and processes the results so that the architectural state updates according to the original program order.

When a micro-op completes and writes its result, it is retired. Up to three micro-ops may be retired per cycle. The Reorder Buffer (ROB) is the unit in the processor which buffers completed micro-ops, updates the architectural state in order, and manages the ordering of exceptions. The retirement section also keeps track of branches and sends updated branch target information to the BTB. The BTB then purges pre-fetched traces that are no longer needed.

2.2.3 Intel® Core™ Microarchitecture

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multi-threaded workloads:

- **Intel® Wide Dynamic Execution** enable each processor core to fetch, dispatch, execute in high bandwidths to support retirement of up to four instructions per cycle.
 - Fourteen-stage efficient pipeline
 - Three arithmetic logical units
 - Four decoders to decode up to five instruction per cycle
 - Macro-fusion and micro-fusion to improve front-end throughput
 - Peak issue rate of dispatching up to six micro-ops per cycle
 - Peak retirement bandwidth of up to 4 micro-ops per cycle
 - Advanced branch prediction
 - Stack pointer tracker to improve efficiency of executing function/procedure entries and exits
- **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, and optimal performance and flexibility for single-threaded and multi-threaded applications.

- Large second level cache up to 4 MB and 16-way associativity
- Optimized for multicore and single-threaded execution environments
- 256 bit internal data path to improve bandwidth from L2 to first-level data cache
- **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution.
 - Hardware prefetchers to reduce effective latency of second-level cache misses
 - Hardware prefetchers to reduce effective latency of first-level data cache misses
 - Memory disambiguation to improve efficiency of speculative execution engine
- **Intel® Advanced Digital Media Boost** improves most 128-bit SIMD instruction with single-cycle throughput and floating-point operations.
 - Single-cycle throughput of most 128-bit SIMD instructions
 - Up to eight floating-point operation per cycle
 - Three issue ports available to dispatching SIMD instructions for execution

Intel Core 2 Extreme, Intel Core 2 Duo processors and Intel Xeon processor 5100 series implement two processor cores based on the Intel Core microarchitecture, the functionality of the subsystems in each core are depicted in Figure 2-3.

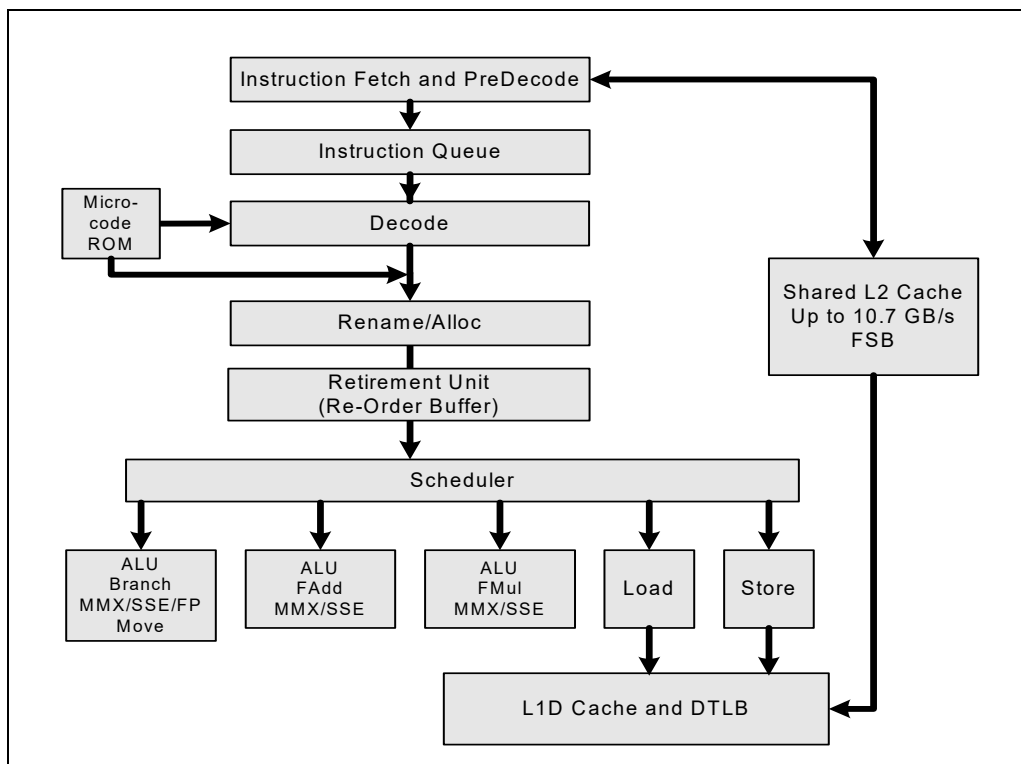


Figure 2-3. The Intel Core Microarchitecture Pipeline Functionality

2.2.3.1 The Front End

The front end of Intel Core microarchitecture provides several enhancements to feed the Intel Wide Dynamic Execution engine:

- Instruction fetch unit prefetches instructions into an instruction queue to maintain steady supply of instruction to the decode units.
- Four-wide decode unit can decode 4 instructions per cycle or 5 instructions per cycle with Macrofusion.

- Macrofusion fuses common sequence of two instructions as one decoded instruction (micro-ops) to increase decoding throughput.
- Microfusion fuses common sequence of two micro-ops as one micro-ops to improve retirement throughput.
- Instruction queue provides caching of short loops to improve efficiency.
- Stack pointer tracker improves efficiency of executing procedure/function entries and exits.
- Branch prediction unit employs dedicated hardware to handle different types of branches for improved branch prediction.
- Advanced branch prediction algorithm directs instruction fetch unit to fetch instructions likely in the architectural code path for decoding.

2.2.3.2 Execution Core

The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order to increase the overall rate of instructions executed per cycle (IPC). The execution core employs the following feature to improve execution throughput and efficiency:

- Up to six micro-ops can be dispatched to execute per cycle
- Up to four instructions can be retired per cycle
- Three full arithmetic logical units
- SIMD instructions can be dispatched through three issue ports
- Most SIMD instructions have 1-cycle throughput (including 128-bit SIMD instructions)
- Up to eight floating-point operation per cycle
- Many long-latency computation operation are pipelined in hardware to increase overall throughput
- Reduced exposure to data access delays using Intel Smart Memory Access

2.2.4 Intel® Atom™ Microarchitecture

Intel Atom microarchitecture maximizes power-efficient performance for single-threaded and multi-threaded workloads by providing:

- **Advanced Micro-Ops Execution**
 - Single-micro-op instruction execution from decode to retirement, including instructions with register-only, load, and store semantics.
 - Sixteen-stage, in-order pipeline optimized for throughput and reduced power consumption.
 - Dual pipelines to enable decode, issue, execution and retirement of two instructions per cycle.
 - Advanced stack pointer to improve efficiency of executing function entry/returns.
- **Intel® Smart Cache**
 - Second level cache is 512 KB and 8-way associativity.
 - Optimized for multi-threaded and single-threaded execution environments
 - 256 bit internal data path between L2 and L1 data cache improves high bandwidth.
- **Efficient Memory Access**
 - Efficient hardware prefetchers to L1 and L2, speculatively loading data likely to be requested by processor to reduce cache miss impact.
- **Intel® Digital Media Boost**
 - Two issue ports for dispatching SIMD instructions to execution units.
 - Single-cycle throughput for most 128-bit integer SIMD instructions
 - Up to six floating-point operations per cycle
 - Up to two 128-bit SIMD integer operations per cycle

- Safe Instruction Recognition (SIR) to allow long-latency floating-point operations to retire out of order with respect to integer instructions.

2.2.5 Intel® Microarchitecture Code Name Nehalem

Intel microarchitecture code name Nehalem provides the foundation for many innovative features of Intel Core i7 processors. It builds on the success of 45 nm Intel Core microarchitecture and provides the following feature enhancements:

- **Enhanced processor core**
 - Improved branch prediction and recovery from misprediction.
 - Enhanced loop streaming to improve front end performance and reduce power consumption.
 - Deeper buffering in out-of-order engine to extract parallelism.
 - Enhanced execution units to provide acceleration in CRC, string/text processing and data shuffling.
- **Smart Memory Access**
 - Integrated memory controller provides low-latency access to system memory and scalable memory bandwidth
 - New cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic
 - Two level TLBs and increased TLB size.
 - Fast unaligned memory access.
- **HyperThreading Technology**
 - Provides two hardware threads (logical processors) per core.
 - Takes advantage of 4-wide execution engine, large L3, and massive memory bandwidth.
- **Dedicated Power management Innovations**
 - Integrated microcontroller with optimized embedded firmware to manage power consumption.
 - Embedded real-time sensors for temperature, current, and power.
 - Integrated power gate to turn off/on per-core power consumption
 - Versatility to reduce power consumption of memory, link subsystems.

2.2.6 Intel® Microarchitecture Code Name Sandy Bridge

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

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

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

For additional information on Intel® Advanced Vector Extensions (AVX), see Section 5.13, “Intel® Advanced Vector Extensions (Intel® AVX)” and Chapter 14, “Programming with AVX, FMA and AVX2” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

2.2.7 SIMD Instructions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

See also:

- Section 5.4, “MMX™ Instructions,” and Chapter 9, “Programming with Intel® MMX™ Technology”
- Section 5.5, “SSE Instructions,” and Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE)”
- Section 5.6, “SSE2 Instructions,” and Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)”
- Section 5.7, “SSE3 Instructions”, Section 5.8, “Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions”, Section 5.9, “SSE4 Instructions”, and Chapter 12, “Programming with Intel® SSE3, SSSE3, Intel® SSE4 and Intel® AESNI”

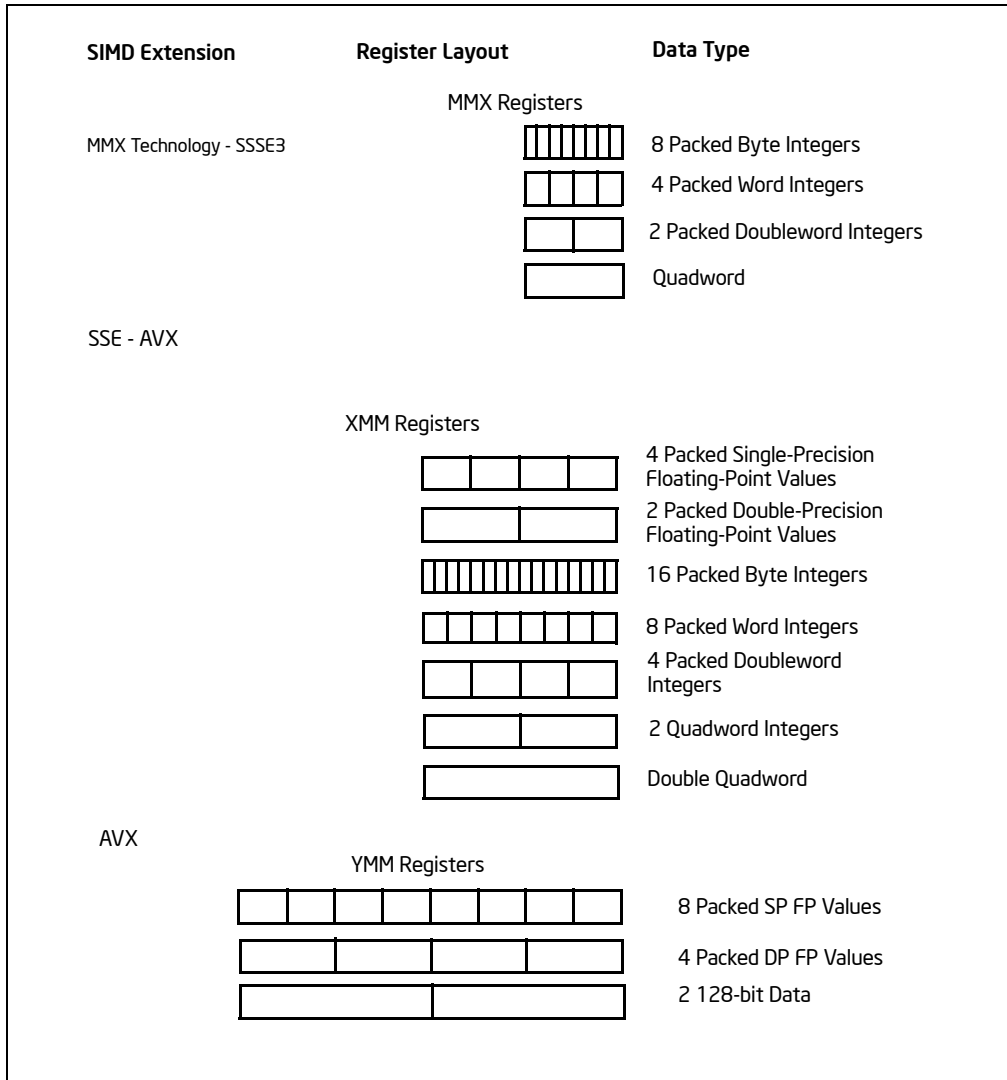


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

2.2.8 Intel® Hyper-Threading Technology

Intel Hyper-Threading Technology (Intel HT Technology) was developed to improve the performance of IA-32 processors when executing multi-threaded operating system and application code or single-threaded applications under multi-tasking environments. The technology enables a single physical processor to execute two or more separate code streams (threads) concurrently using shared execution resources.

Intel HT Technology is one form of hardware multi-threading capability in IA-32 processor families. It differs from multi-processor capability using separate physically distinct packages with each physical processor package mated with a physical socket. Intel HT Technology provides hardware multi-threading capability with a single physical package by using shared execution resources in a processor core.

Architecturally, an IA-32 processor that supports Intel HT Technology consists of two or more logical processors, each of which has its own IA-32 architectural state. Each logical processor consists of a full set of IA-32 data registers, segment registers, control registers, debug registers, and most of the MSRs. Each also has its own advanced programmable interrupt controller (APIC).

Figure 2-5 shows a comparison of a processor that supports Intel HT Technology (implemented with two logical processors) and a traditional dual processor system.

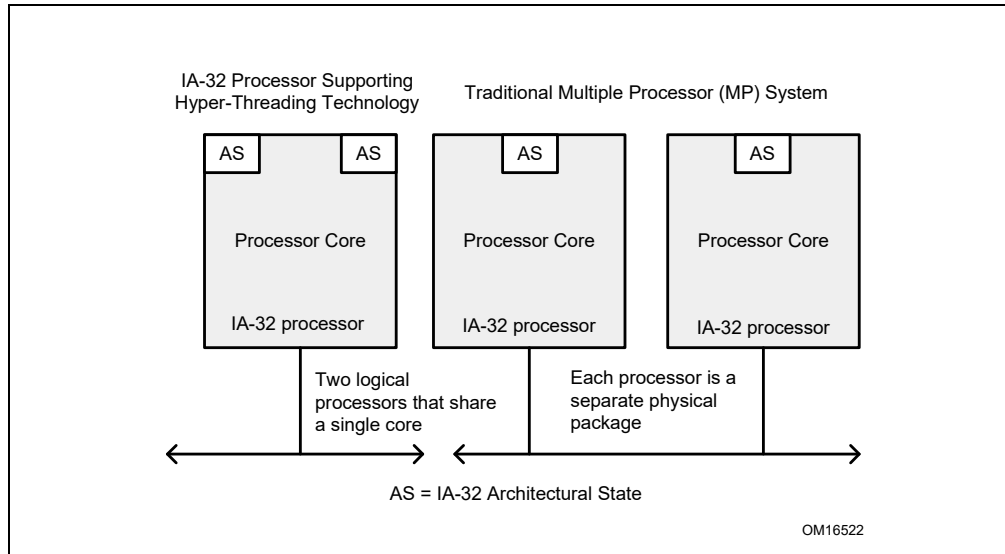


Figure 2-5. Comparison of an IA-32 Processor Supporting Hyper-Threading Technology and a Traditional Dual Processor System

Unlike a traditional MP system configuration that uses two or more separate physical IA-32 processors, the logical processors in an IA-32 processor supporting Intel HT Technology share the core resources of the physical processor. This includes the execution engine and the system bus interface. After power up and initialization, each logical processor can be independently directed to execute a specified thread, interrupted, or halted.

Intel HT Technology leverages the process and thread-level parallelism found in contemporary operating systems and high-performance applications by providing two or more logical processors on a single chip. This configuration allows two or more threads¹ to be executed simultaneously on each a physical processor. Each logical processor executes instructions from an application thread using the resources in the processor core. The core executes these threads concurrently, using out-of-order instruction scheduling to maximize the use of execution units during each clock cycle.

2.2.8.1 Some Implementation Notes

All Intel HT Technology configurations require:

- A processor that supports Intel HT Technology
- A chipset and BIOS that utilize the technology
- Operating system optimizations

See http://www.intel.com/products/ht/hyperthreading_more.htm for information.

At the firmware (BIOS) level, the basic procedures to initialize the logical processors in a processor supporting Intel HT Technology are the same as those for a traditional DP or MP platform. The mechanisms that are described in the *Multiprocessor Specification, Version 1.4* to power-up and initialize physical processors in an MP system also apply to logical processors in a processor that supports Intel HT Technology.

An operating system designed to run on a traditional DP or MP platform may use CPUID to determine the presence of hardware multi-threading support feature and the number of logical processors they provide.

Although existing operating system and application code should run correctly on a processor that supports Intel HT Technology, some code modifications are recommended to get the optimum benefit. These modifications are discussed in Chapter 7, "Multiple-Processor Management," *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. In the remainder of this document, the term "thread" will be used as a general term for the terms "process" and "thread."

2.2.9 Multi-Core Technology

Multi-core technology is another form of hardware multi-threading capability in IA-32 processor families. Multi-core technology enhances hardware multi-threading capability by providing two or more execution cores in a physical package.

The Intel Pentium processor Extreme Edition is the first member in the IA-32 processor family to introduce multi-core technology. The processor provides hardware multi-threading support with both two processor cores and Intel Hyper-Threading Technology. This means that the Intel Pentium processor Extreme Edition provides four logical processors in a physical package (two logical processors for each processor core). The Dual-Core Intel Xeon processor features multi-core, Intel Hyper-Threading Technology and supports multi-processor platforms.

The Intel Pentium D processor also features multi-core technology. This processor provides hardware multi-threading support with two processor cores but does not offer Intel Hyper-Threading Technology. This means that the Intel Pentium D processor provides two logical processors in a physical package, with each logical processor owning the complete execution resources of a processor core.

The Intel Core 2 processor family, Intel Xeon processor 3000 series, Intel Xeon processor 5100 series, and Intel Core Duo processor offer power-efficient multi-core technology. The processor contains two cores that share a smart second level cache. The Level 2 cache enables efficient data sharing between two cores to reduce memory traffic to the system bus.

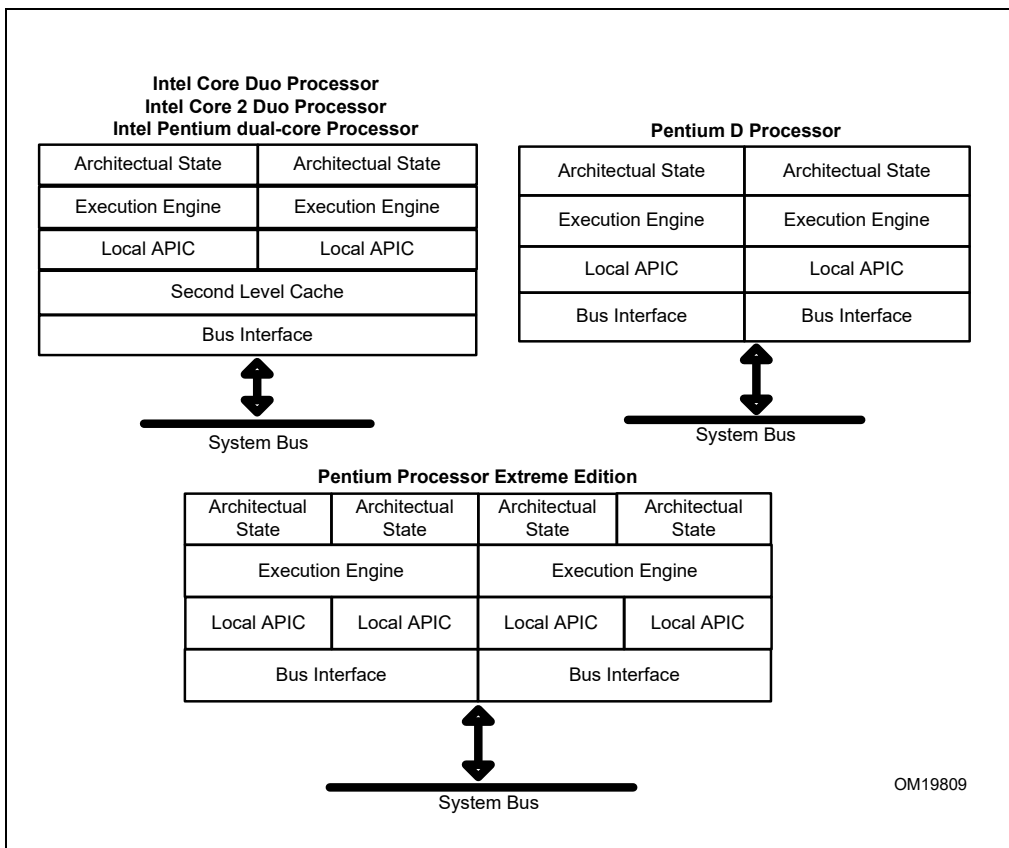


Figure 2-6. Intel 64 and IA-32 Processors that Support Dual-Core

The Pentium® dual-core processor is based on the same technology as the Intel Core 2 Duo processor family.

The Intel Xeon processor 7300, 5300 and 3200 series, Intel Core 2 Extreme Quad-Core processor, and Intel Core 2 Quad processors support Intel quad-core technology. The Quad-core Intel Xeon processors and the Quad-Core Intel Core 2 processor family are also in Figure 2-7.

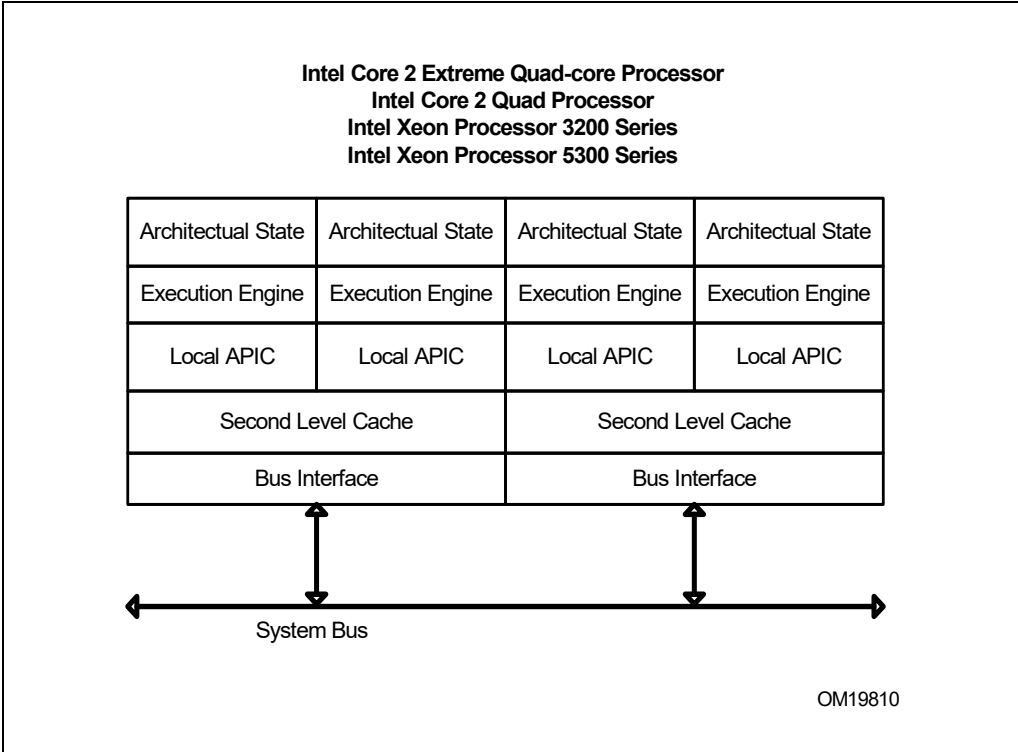


Figure 2-7. Intel 64 Processors that Support Quad-Core

Intel Core i7 processors support Intel quad-core technology, Intel HyperThreading Technology, provides Intel QuickPath interconnect link to the chipset and have integrated memory controller supporting three channel to DDR3 memory.

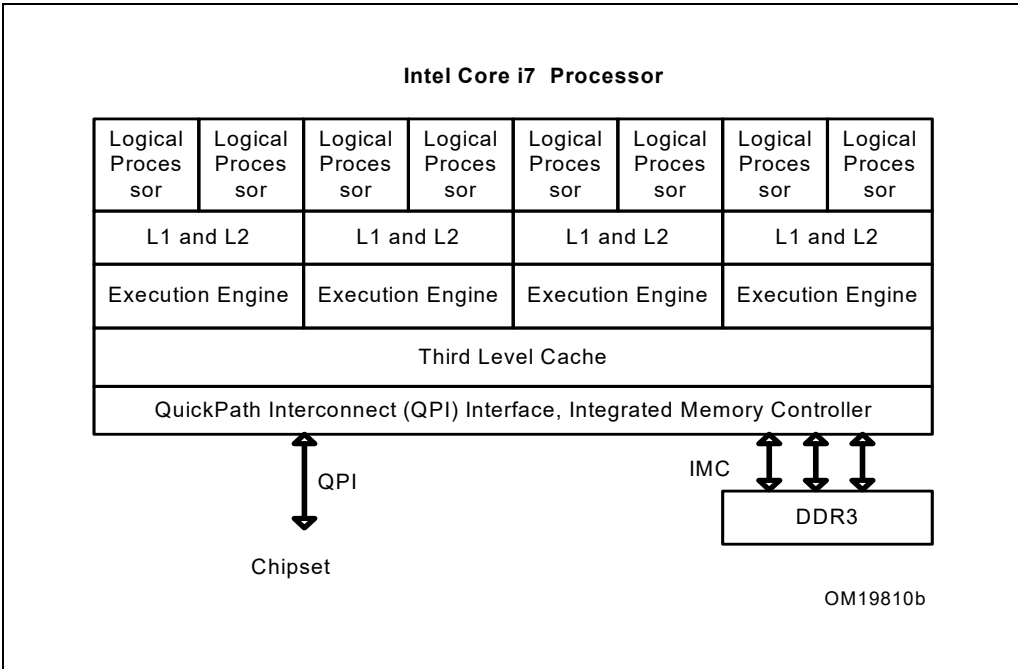


Figure 2-8. Intel Core i7 Processor

2.2.10 Intel® 64 Architecture

Intel 64 architecture increases the linear address space for software to 64 bits and supports physical address space up to 52 bits. The technology also introduces a new operating mode referred to as IA-32e mode.

IA-32e mode operates in one of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit address space.

In the 64-bit mode, applications may access:

- 64-bit flat linear addressing
- 8 additional general-purpose registers (GPRs)
- 8 additional registers for streaming SIMD extensions (SSE, SSE2, SSE3 and SSSE3)
- 64-bit-wide GPRs and instruction pointers
- uniform byte-register addressing
- fast interrupt-prioritization mechanism
- a new instruction-pointer relative-addressing mode

An Intel 64 architecture processor supports existing IA-32 software because it is able to run all non-64-bit legacy modes supported by IA-32 architecture. Most existing IA-32 applications also run in compatibility mode.

2.2.11 Intel® Virtualization Technology (Intel® VT)

Intel® Virtualization Technology for Intel 64 and IA-32 architectures provide extensions that support virtualization. The extensions are referred to as Virtual Machine Extensions (VMX). An Intel 64 or IA-32 platform with VMX can function as multiple virtual systems (or virtual machines). Each virtual machine can run operating systems and applications in separate partitions.

VMX also provides programming interface for a new layer of system software (called the Virtual Machine Monitor (VMM)) used to manage the operation of virtual machines. Information on VMX and on the programming of VMMs is in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

Intel Core i7 processor provides the following enhancements to Intel Virtualization Technology:

- Virtual processor ID (VPID) to reduce the cost of VMM managing transitions.
- Extended page table (EPT) to reduce the number of transitions for VMM to manage memory virtualization.
- Reduced latency of VM transitions.

2.3 INTEL® 64 AND IA-32 PROCESSOR GENERATIONS

In the mid-1960s, Intel cofounder and Chairman Emeritus Gordon Moore had this observation: "... the number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years." Over the past three and half decades, this prediction known as "Moore's Law" has continued to hold true.

The computing power and the complexity (or roughly, the number of transistors per processor) of Intel architecture processors has grown in close relation to Moore's law. By taking advantage of new process technology and new microarchitecture designs, each new generation of IA-32 processors has demonstrated frequency-scaling headroom and new performance levels over the previous generation processors.

The key features of the Intel Pentium 4 processor, Intel Xeon processor, Intel Xeon processor MP, Pentium III processor, and Pentium III Xeon processor with advanced transfer cache are shown in Table 2-1. Older generation IA-32 processors, which do not employ on-die Level 2 cache, are shown in Table 2-2.

Table 2-1. Key Features of Most Recent IA-32 Processors

Intel Processor	Date Introduced	Micro-architecture	Top-Bin Clock Frequency at Introduction	Transistors	Register Sizes ¹	System Bus Bandwidth	Max. Extern. Addr. Space	On-Die Caches ²
Intel Pentium M Processor 755 ³	2004	Intel Pentium M Processor	2.00 GHz	140 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	4 GB	L1: 64 KB L2: 2 MB
Intel Core Duo Processor T2600 ³	2006	Improved Intel Pentium M Processor Microarchitecture; Dual Core; Intel Smart Cache, Advanced Thermal Manager	2.16 GHz	152M	GP: 32 FPU: 80 MMX: 64 XMM: 128	5.3 GB/s	4 GB	L1: 64 KB L2: 2 MB (2MB Total)
Intel Atom Processor Z5xx series	2008	Intel Atom Microarchitecture; Intel Virtualization Technology.	1.86 GHz - 800 MHz	47M	GP: 32 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	4 GB	L1: 56 KB ⁴ L2: 512KB

NOTES:

1. The register size and external data bus size are given in bits.
2. First level cache is denoted using the abbreviation L1, 2nd level cache is denoted as L2. The size of L1 includes the first-level data cache and the instruction cache where applicable, but does not include the trace cache.
3. Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families.
See http://www.intel.com/products/processor_number for details.
4. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
64-bit Intel Xeon Processor with 800 MHz System Bus	2004	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.60 GHz	125 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 1 MB L2
64-bit Intel Xeon Processor MP with 8MB L3	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.33 GHz	675M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	5.3 GB/s ¹	1024 GB (1 TB)	12K μ op Execution Trace Cache; 16 KB L1; 1 MB L2, 8 MB L3

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.73 GHz	164 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 2 MB L2
Intel Pentium Processor Extreme Edition 840	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core ²	3.20 GHz	230 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 1MB L2 (2MB Total)
Dual-Core Intel Xeon Processor 7041	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core ³	3.00 GHz	321M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 2MB L2 (4MB Total)
Intel Pentium 4 Processor 672	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.80 GHz	164 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 2MB L2
Intel Pentium Processor Extreme Edition 955	2006	Intel NetBurst Microarchitecture; Intel 64 Architecture; Dual Core; Intel Virtualization Technology.	3.46 GHz	376M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	12K μ op Execution Trace Cache; 16 KB L1; 2MB L2 (4MB Total)
Intel Core 2 Extreme Processor X6800	2006	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.	2.93 GHz	291M	GP: 32,64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
Intel Xeon Processor 5160	2006	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.	3.00 GHz	291M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Intel Xeon Processor 7140	2006	Intel NetBurst Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.	3.40 GHz	1.3 B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	12.8 GB/s	64 GB	L1: 64 KB L2: 1MB (2MB Total) L3: 16 MB (16MB Total)
Intel Core 2 Extreme Processor QX6700	2006	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.	2.66 GHz	582M	GP: 32,64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Quad-core Intel Xeon Processor 5355	2006	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.	2.66 GHz	582 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	256 GB	L1: 64 KB L2: 4MB (8 MB Total)
Intel Core 2 Duo Processor E6850	2007	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology; Intel Trusted Execution Technology	3.00 GHz	291 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Intel Xeon Processor 7350	2007	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.	2.93 GHz	582 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	1024 GB	L1: 64 KB L2: 4MB (8MB Total)

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
Intel Xeon Processor 5472	2007	Enhanced Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.	3.00 GHz	820 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	12.8 GB/s	256 GB	L1: 64 KB L2: 6MB (12MB Total)
Intel Atom Processor	2008	Intel Atom Microarchitecture; Intel 64 Architecture; Intel Virtualization Technology.	2.0 - 1.60 GHz	47 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	Up to 64GB	L1: 56 KB ⁴ L2: 512KB
Intel Xeon Processor 7460	2008	Enhanced Intel Core Microarchitecture; Six Cores; Intel 64 Architecture; Intel Virtualization Technology.	2.67 GHz	1.9 B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	1024 GB	L1: 64 KB L2: 3MB (9MB Total) L3: 16MB
Intel Atom Processor 330	2008	Intel Atom Microarchitecture; Intel 64 Architecture; Dual core; Intel Virtualization Technology.	1.60 GHz	94 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	Up to 64GB	L1: 56 KB ⁵ L2: 512KB (1MB Total)
Intel Core i7-965 Processor Extreme Edition	2008	Intel microarchitecture code name Nehalem; Quadcore; HyperThreading Technology; Intel QPI; Intel 64 Architecture; Intel Virtualization Technology.	3.20 GHz	731 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 25 GB/s	64 GB	L1: 64 KB L2: 256KB L3: 8MB

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
Intel Core i7-620M Processor	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Dualcore; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology., Integrated graphics	2.66 GHz	383 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128		64 GB	L1: 64 KB L2: 256KB L3: 4MB
Intel Xeon-Processor 5680	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Six core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.33 GHz	1.1B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; 32 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 12MB
Intel Xeon-Processor 7560	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Nehalem; Eight core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	2.26 GHz	2.3B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 76 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 24MB
Intel Core i7-2600K Processor	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Sandy Bridge; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology., Processor graphics, Quicksync Video	3.40 GHz	995M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	64 GB	L1: 64 KB L2: 256KB L3: 8MB

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

Intel Processor	Date Introduced	Micro-architecture	Highest Processor Base Frequency at Introduction	Transistors	Register Sizes	System Bus/QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches
Intel Xeon-Processor E3-1280	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Sandy Bridge; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.50 GHz		GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 8MB
Intel Xeon-Processor E7-8870	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Ten core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	2.40 GHz	2.2B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 102 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 30MB

NOTES:

1. The 64-bit Intel Xeon Processor MP with an 8-MByte L3 supports a multi-processor platform with a dual system bus; this creates a platform bandwidth with 10.6 GBytes.
2. In Intel Pentium Processor Extreme Edition 840, the size of on-die cache is listed for each core. The total size of L2 in the physical package is 2 MBytes.
3. In Dual-Core Intel Xeon Processor 7041, the size of on-die cache is listed for each core. The total size of L2 in the physical package is 4 MBytes.
4. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.
5. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.

Table 2-3. Key Features of Previous Generations of IA-32 Processors

Intel Processor	Date Introduced	Max. Clock Frequency/ Technology at Introduction	Transistors	Register Sizes ¹	Ext. Data Bus Size ²	Max. Extern. Addr. Space	Caches
8086	1978	8 MHz	29 K	16 GP	16	1 MB	None
Intel 286	1982	12.5 MHz	134 K	16 GP	16	16 MB	Note 3
Intel386 DX Processor	1985	20 MHz	275 K	32 GP	32	4 GB	Note 3
Intel486 DX Processor	1989	25 MHz	1.2 M	32 GP 80 FPU	32	4 GB	L1: 8 KB
Pentium Processor	1993	60 MHz	3.1 M	32 GP 80 FPU	64	4 GB	L1:16 KB
Pentium Pro Processor	1995	200 MHz	5.5 M	32 GP 80 FPU	64	64 GB	L1: 16 KB L2: 256 KB or 512 KB
Pentium II Processor	1997	266 MHz	7 M	32 GP 80 FPU 64 MMX	64	64 GB	L1: 32 KB L2: 256 KB or 512 KB
Pentium III Processor	1999	500 MHz	8.2 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 32 KB L2: 512 KB
Pentium III and Pentium III Xeon Processors	1999	700 MHz	28 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 32 KB L2: 256 KB
Pentium 4 Processor	2000	1.50 GHz, Intel NetBurst Microarchitecture	42 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	12K μ op Execution Trace Cache; L1: 8KB L2: 256 KB
Intel Xeon Processor	2001	1.70 GHz, Intel NetBurst Microarchitecture	42 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	12K μ op Execution Trace Cache; L1: 8KB L2: 512KB
Intel Xeon Processor	2002	2.20 GHz, Intel NetBurst Microarchitecture, HyperThreading Technology	55 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	12K μ op Execution Trace Cache; L1: 8KB L2: 512KB
Pentium M Processor	2003	1.60 GHz, Intel NetBurst Microarchitecture	77 M	32 GP 80 FPU 64 MMX 128 XMM	64	4 GB	L1: 64KB L2: 1 MB

Table 2-3. Key Features of Previous Generations of IA-32 Processors (Contd.)

Intel Pentium 4 Processor Supporting Hyper-Threading Technology at 90 nm process	2004	3.40 GHz, Intel NetBurst Microarchitecture, HyperThreading Technology	125 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	12K μ op Execution Trace Cache; L1: 16KB L2: 1 MB
--	------	---	-------	--------------------------------------	----	-------	---

NOTE:

1. The register size and external data bus size are given in bits. Note also that each 32-bit general-purpose (GP) registers can be addressed as an 8- or a 16-bit data registers in all of the processors.
2. Internal data paths are 2 to 4 times wider than the external data bus for each processor.

2.4 PROPOSED REMOVAL OF INTEL INSTRUCTION SET ARCHITECTURE AND FEATURES FROM UPCOMING PRODUCTS

This section lists Intel Instruction Set Architecture (ISA) and features that Intel plans to remove from select products starting from a specific year.

Table 2-4. Proposed Intel ISA and Features Removal List

Intel ISA/Feature	Year of Removal
NA	NA

2.5 INTEL INSTRUCTION SET ARCHITECTURE AND FEATURES REMOVED

This section lists Intel ISA and features that Intel has already removed for select upcoming products. All sections relevant to the removed features will be identified as such and may be moved to an archived section in future Intel® 64 and IA-32 Architectures Software Developer's Manual releases.

Table 2-5. Intel ISA and Features Removal List

Intel ISA/Feature	Year of Removal
Intel® Memory Protection Extensions (Intel® MPX)	2019 onwards
MSR_TEST_CTRL, bit 31 (MSR address 33H)	2019 onwards
Hardware Lock Elision (HLE)	2019 onwards

2. Updates to Chapter 3, Volume 1

Change bars and green text show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

Changes to this chapter: Typo corrections.

This chapter describes the basic execution environment of an Intel 64 or IA-32 processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The execution environment described here includes memory (the address space), general-purpose data registers, segment registers, the flag register, and the instruction pointer register.

3.1 MODES OF OPERATION

The IA-32 architecture supports three basic operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

- **Protected mode** — This mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute “real-address mode” 8086 software in a protected, multi-tasking environment. This feature is called **virtual-8086 mode**, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.
- **Real-address mode** — This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.
- **System management mode (SMM)** — This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC).

In SMM, the processor switches to a separate address space while saving the basic 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 system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

3.1.1 Intel® 64 Architecture

Intel 64 architecture adds IA-32e mode. IA-32e mode has two sub-modes. These are:

- **Compatibility mode (sub-mode of IA-32e mode)** — Compatibility mode permits most legacy 16-bit and 32-bit applications to run without re-compilation under a 64-bit operating system. For brevity, the compatibility sub-mode is referred to as compatibility mode in IA-32 architecture. The execution environment of compatibility mode is the same as described in Section 3.2. Compatibility mode also supports all of the privilege levels that are supported in 64-bit and protected modes. Legacy applications that run in Virtual 8086 mode or use hardware task management will not work in this mode.

Compatibility mode is enabled by the operating system (OS) on a code segment basis. This means that a single 64-bit OS can support 64-bit applications running in 64-bit mode and support legacy 32-bit applications (not recompiled for 64-bits) running in compatibility mode.

Compatibility mode is similar to 32-bit protected mode. Applications access only the first 4 GByte of linear-address space. Compatibility mode uses 16-bit and 32-bit address and operand sizes. Like protected mode, this mode allows applications to access physical memory greater than 4 GByte using PAE (Physical Address Extensions).

- **64-bit mode (sub-mode of IA-32e mode)** — This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space. For brevity, the 64-bit sub-mode is referred to as 64-bit mode in IA-32 architecture.

64-bit mode extends the number of general purpose registers and SIMD extension registers from 8 to 16. General purpose registers are widened to 64 bits. The mode also introduces a new opcode prefix (REX) to access the register extensions. See Section 3.2.1 for a detailed description.

64-bit mode is enabled by the operating system on a code-segment basis. Its default address size is 64 bits and its default operand size is 32 bits. The default operand size can be overridden on an instruction-by-instruction basis using a REX opcode prefix in conjunction with an operand size override prefix.

REX prefixes allow a 64-bit operand to be specified when operating in 64-bit mode. By using this mechanism, many existing instructions have been promoted to allow the use of 64-bit registers and 64-bit addresses.

3.2 OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an IA-32 processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (described briefly in the following paragraphs and shown in Figure 3-1) make up the basic execution environment for an IA-32 processor.

An Intel 64 processor supports the basic execution environment of an IA-32 processor, and a similar environment under IA-32e mode that can execute 64-bit programs (64-bit sub-mode) and 32-bit programs (compatibility sub-mode).

The basic execution environment is used jointly by the application programs and the operating system or executive running on the processor.

- **Address space** — Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes (2^{32} bytes) and a physical address space of up to 64 GBytes (2^{36} bytes). See Section 3.3.6, “Extended Physical Addressing in Protected Mode,” for more information about addressing an address space greater than 4 GBytes.
- **Basic program execution registers** — The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory. See Section 3.4, “Basic Program Execution Registers,” for more information about these registers.
- **x87 FPU registers** — The eight x87 FPU data registers, the x87 FPU control register, the status register, the x87 FPU instruction pointer register, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single-precision, double-precision, and double extended-precision floating-point values, word integers, doubleword integers, quadword integers, and binary coded decimal (BCD) values. See Section 8.1, “x87 FPU Execution Environment,” for more information about these registers.
- **MMX registers** — The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers. See Section 9.2, “The MMX Technology Programming Environment,” for more information about these registers.
- **XMM registers** — The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers. See Section 10.2, “SSE Programming Environment,” for more information about these registers.
- **YMM registers** — The YMM data registers support execution of 256-bit SIMD operations on 256-bit packed single-precision and double-precision floating-point values and on 256-bit packed byte, word, doubleword, and quadword integers.
- **Bounds registers** — Each of the BND0-BND3 register stores the lower and upper **bounds** (64 bits each) associated with the pointer to a memory buffer. They support execution of the Intel MPX instructions.
- **BNDCFGU and BNDSTATUS**— BNDCFGU configures user mode MPX operations on bounds checking. BNDSTATUS provides additional information on the #BR caused by an MPX operation.

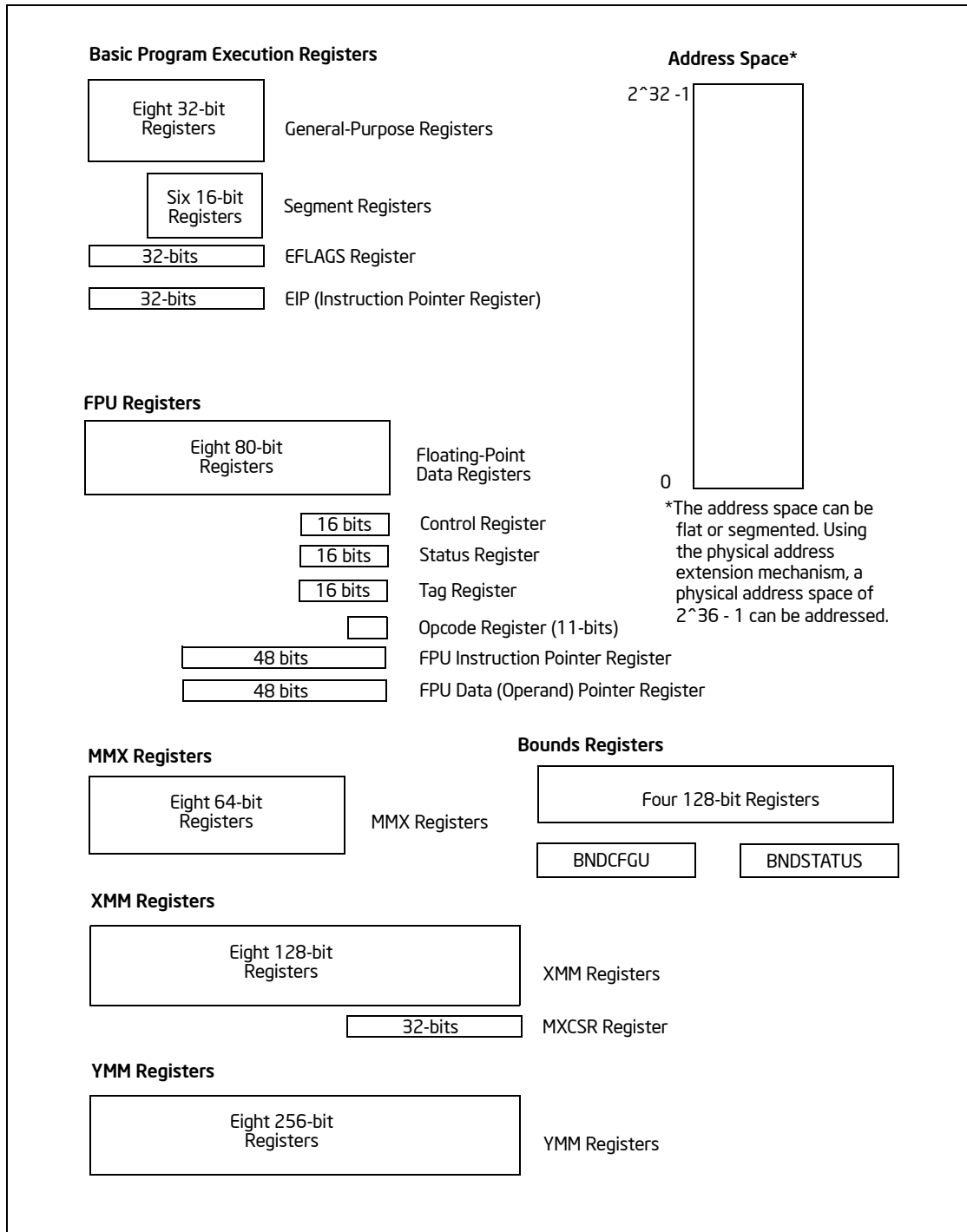


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

- **Stack** — To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment. The stack (not shown in Figure 3-1) is located in memory. See Section 6.2, “Stacks,” for more information about stack structure.

In addition to the resources provided in the basic execution environment, the IA-32 architecture provides the following resources as part of its system-level architecture. They provide extensive support for operating-system and system-development software. Except for the I/O ports, the system resources are described in detail in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A & 3B*.

- **I/O ports** — The IA-32 architecture supports a transfers of data to and from input/output (I/O) ports. See Chapter 19, “Input/Output,” in this volume.
- **Control registers** — The five control registers (CR0 through CR4) determine the operating mode of the processor and the characteristics of the currently executing task. See Chapter 2, “System Architecture Overview,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- **Memory management registers** — The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management. See Chapter 2, “System Architecture Overview,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- **Debug registers** — The debug registers (DR0 through DR7) control and allow monitoring of the processor’s debugging operations. See in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.
- **Memory type range registers (MTRRs)** — The MTRRs are used to assign memory types to regions of memory. See the sections on MTRRs in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A & 3B*.
- **Model-specific registers (MSRs)** — The processor provides a variety of model-specific registers that are used to control and report on processor performance. Virtually all MSRs handle system related functions and are not accessible to an application program. One exception to this rule is the time-stamp counter. The MSRs are described in Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.
- **Machine check registers** — The machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors. See Chapter 15, “Machine-Check Architecture,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- **Performance monitoring counters** — The performance monitoring counters allow processor performance events to be monitored. See Chapter 18, “Performance Monitoring,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

The remainder of this chapter describes the organization of memory and the address space, the basic program execution registers, and addressing modes. Refer to the following chapters in this volume for descriptions of the other program execution resources shown in Figure 3-1:

- **x87 FPU registers** — See Chapter 8, “Programming with the x87 FPU.”
- **MMX Registers** — See Chapter 9, “Programming with Intel® MMX™ Technology.”
- **XMM registers** — See Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE),” Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2),” and Chapter 12, “Programming with Intel® SSE3, SSSE3, Intel® SSE4 and Intel® AESNI.”
- **YMM registers** — See Chapter 14, “Programming with AVX, FMA and AVX2”.
- **BND registers, BNDCFGU, BNDSTATUS** — See Chapter 13, “Managing State Using the XSAVE Feature Set,” and Chapter 17, “Intel® MPX”.
- **Stack implementation and procedure calls** — See Chapter 6, “Procedure Calls, Interrupts, and Exceptions.”

3.2.1 64-Bit Mode Execution Environment

The execution environment for 64-bit mode is similar to that described in Section 3.2. The following paragraphs describe the differences that apply.

- **Address space** — A task or program running in 64-bit mode on an IA-32 processor can address linear address space of up to 2^{64} bytes (subject to the canonical addressing requirement described in Section 3.3.7.1) and physical address space of up to 2^{52} bytes. Software can query CPUID for the physical address size supported by a processor.
- **Basic program execution registers** — The number of general-purpose registers (GPRs) available is 16. GPRs are 64-bits wide and they support operations on byte, word, doubleword and quadword integers. Accessing byte registers is done uniformly to the lowest 8 bits. The instruction pointer register becomes 64 bits. The EFLAGS register is extended to 64 bits wide, and is referred to as the RFLAGS register. The upper 32 bits of RFLAGS is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS. See Figure 3-2.
- **XMM registers** — There are 16 XMM data registers for SIMD operations. See Section 10.2, “SSE Programming Environment,” for more information about these registers.
- **YMM registers** — There are 16 YMM data registers for SIMD operations. See Chapter 14, “Programming with AVX, FMA and AVX2” for more information about these registers.
- **BND registers, BNDCFGU, BNDSTATUS** — See Chapter 13, “Managing State Using the XSAVE Feature Set,” and Chapter 17, “Intel® MPX”.
- **Stack** — The stack pointer size is 64 bits. Stack size is not controlled by a bit in the SS descriptor (as it is in non-64-bit modes) nor can the pointer size be overridden by an instruction prefix.
- **Control registers** — Control registers expand to 64 bits. A new control register (the task priority register: CR8 or TPR) has been added. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in this volume.
- **Debug registers** — Debug registers expand to 64 bits. See Chapter 17, “Debug, Branch Profile, TSC, and Quality of Service,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

- **Descriptor table registers** — The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) expand to 10 bytes so that they can hold a full 64-bit base address. The local descriptor table register (LDTR) and the task register (TR) also expand to hold a full 64-bit base address.

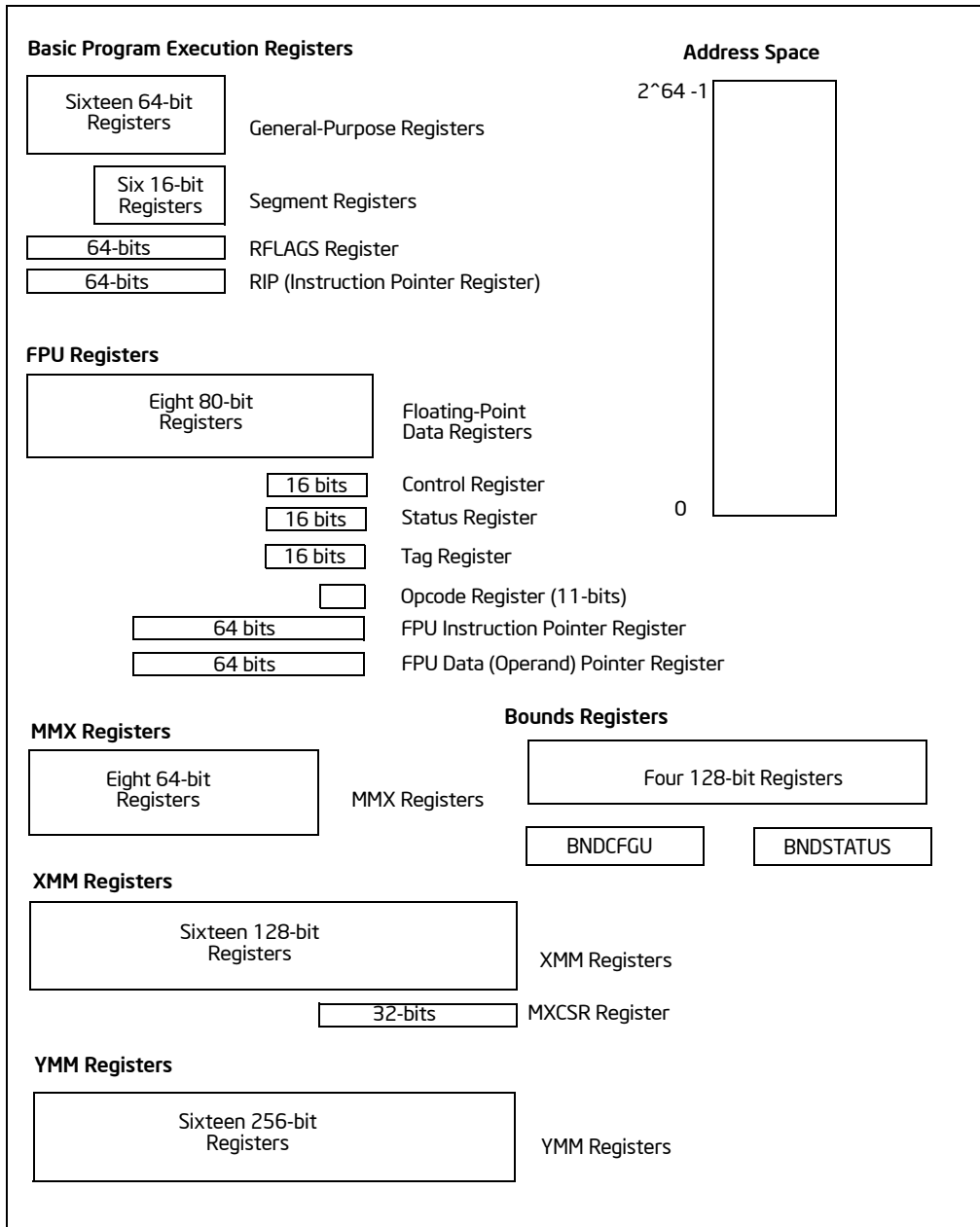


Figure 3-2. 64-Bit Mode Execution Environment

3.3 MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of $2^{36} - 1$ (64 GBytes) if the processor does not support Intel 64 architecture. Intel 64 architecture introduces a set of changes in physical and linear address space; these are described in Section 3.3.3, Section 3.3.4, and Section 3.3.7.

Virtually any operating system or executive designed to work with an IA-32 or Intel 64 processor will use the processor's memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Memory management is described in detail in Chapter 3, "Protected-Mode Memory Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. The following paragraphs describe the basic methods of addressing memory when memory management is used.

3.3.1 IA-32 Memory Models

When employing the processor's memory management facilities, programs do not directly address physical memory. Instead, they access memory using one of three memory models: flat, segmented, or real address mode:

- **Flat memory model** — Memory appears to a program as a single, continuous address space (Figure 3-3). This space is called a **linear address space**. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$ (if not in 64-bit mode). An address for any byte in linear address space is called a **linear address**.
- **Segmented memory model** — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as 2^{32} bytes.

Internally, all the segments that are defined for a system are mapped into the processor's linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

- **Real-address mode memory model** — This is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is 2^{20} bytes.

See also: Chapter 20, "8086 Emulation," *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

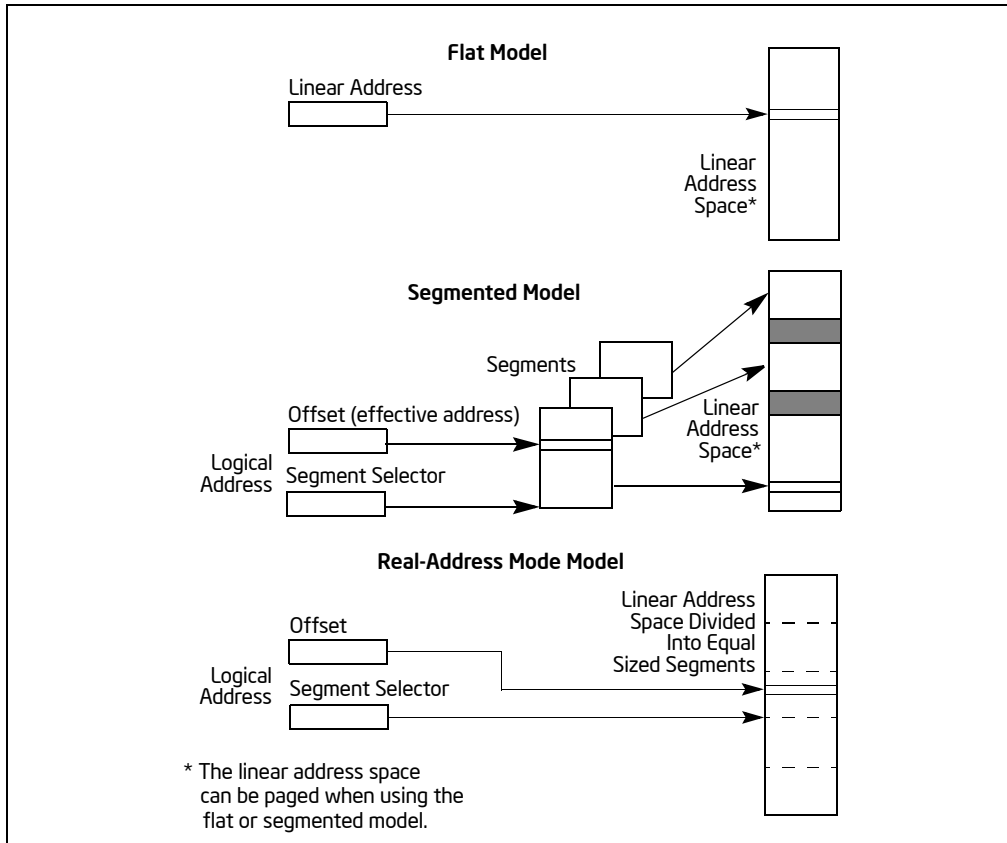


Figure 3-3. Three Memory Management Models

3.3.2 Paging and Virtual Memory

With the flat or the segmented memory model, linear address space is mapped into the processor’s physical address space either directly or through paging. When using direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address. Linear addresses are sent out on the processor’s address lines without translation.

When using the IA-32 architecture’s paging mechanism (paging enabled), linear address space is divided into pages which are mapped to virtual memory. The pages of virtual memory are then mapped as needed into physical memory. When an operating system or executive uses paging, the paging mechanism is transparent to an application program. All that the application sees is linear address space.

In addition, IA-32 architecture’s paging mechanism includes extensions that support:

- Physical Address Extensions (PAE) to address physical address space greater than 4 GBytes.
- Page Size Extensions (PSE) to map linear address to physical address in 4-MBytes pages.

See also: Chapter 3, “Protected-Mode Memory Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

3.3.3 Memory Organization in 64-Bit Mode

Intel 64 architecture supports physical address space greater than 64 GBytes; the actual physical address size of IA-32 processors is implementation specific. In 64-bit mode, there is architectural support for 64-bit linear address space. However, processors supporting Intel 64 architecture may implement less than 64-bits (see Section 3.3.7.1). The linear address space is mapped into the processor physical address space through the PAE paging mechanism.

3.3.4 Modes of Operation vs. Memory Model

When writing code for an IA-32 or Intel 64 processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

- **Protected mode** — When in protected mode, the processor can use any of the memory models described in this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the virtual-8086 mode.) The memory model used depends on the design of the operating system or executive. When multitasking is implemented, individual tasks can use different memory models.
- **Real-address mode** — When in real-address mode, the processor only supports the real-address mode memory model.
- **System management mode** — When in SMM, the processor switches to a separate address space, called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. See Chapter 34, “System Management Mode,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information on the memory model used in SMM.
- **Compatibility mode** — Software that needs to run in compatibility mode should observe the same memory model as those targeted to run in 32-bit protected mode. The effect of segmentation is the same as it is in 32-bit protected mode semantics.
- **64-bit mode** — Segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. Specifically, the processor treats the segment base of CS, DS, ES, and SS as zero in 64-bit mode (this makes a linear address equal an effective address). Segmented and real address modes are not available in 64-bit mode.

3.3.5 32-Bit and 16-Bit Address and Operand Sizes

IA-32 processors in protected mode can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH ($2^{32}-1$); operand sizes are typically 8 bits or 32 bits. With 16-bit address and operand sizes, the maximum linear address or segment offset is FFFFH ($2^{16}-1$); operand sizes are typically 8 bits or 16 bits.

When using 32-bit addressing, a logical address (or far pointer) consists of a 16-bit segment selector and a 32-bit offset; when using 16-bit addressing, an address consists of a 16-bit segment selector and a 16-bit offset.

Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program.

When operating in protected mode, the segment descriptor for the currently executing code segment defines the default address and operand size. A segment descriptor is a system data structure not normally visible to application code. Assembler directives allow the default addressing and operand size to be chosen for a program. The assembler and other tools then set up the segment descriptor for the code segment appropriately.

When operating in real-address mode, the default addressing and operand size is 16 bits. An address-size override can be used in real-address mode to enable 32-bit addressing. However, the maximum allowable 32-bit linear address is still 00FFFFFFH ($2^{20}-1$).

3.3.6 Extended Physical Addressing in Protected Mode

Beginning with P6 family processors, the IA-32 architecture supports addressing of up to 64 GBytes (2^{36} bytes) of physical memory. A program or task could not address locations in this address space directly. Instead, it addresses individual linear address spaces of up to 4 GBytes that mapped to 64-GByte physical address space through a virtual memory management mechanism. Using this mechanism, an operating system can enable a program to switch 4-GByte linear address spaces within 64-GByte physical address space.

The use of extended physical addressing requires the processor to operate in protected mode and the operating system to provide a virtual memory management system. See “36-Bit Physical Addressing Using the PAE Paging Mechanism” in Chapter 3, “Protected-Mode Memory Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

3.3.7 Address Calculations in 64-Bit Mode

In most cases, 64-bit mode uses flat address space for code, data, and stacks. In 64-bit mode (if there is no address-size override), the size of effective address calculations is 64 bits. An effective-address calculation uses a 64-bit base and index registers and sign-extend displacements to 64 bits.

In the flat address space of 64-bit mode, linear addresses are equal to effective addresses because the base address is zero. In the event that FS or GS segments are used with a non-zero base, this rule does not hold. In 64-bit mode, the effective address components are added and the effective address is truncated (See for example the instruction LEA) before adding the full 64-bit segment base. The base is never truncated, regardless of addressing mode in 64-bit mode.

The instruction pointer is extended to 64 bits to support 64-bit code offsets. The 64-bit instruction pointer is called the RIP. Table 3-1 shows the relationship between RIP, EIP, and IP.

Table 3-1. Instruction Pointer Sizes

	Bits 63:32	Bits 31:16	Bits 15:0
16-bit instruction pointer	Not Modified		IP
32-bit instruction pointer	Zero Extension	EIP	
64-bit instruction pointer	RIP		

Generally, displacements and immediates in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and sign-extended during effective-address calculations. In 64-bit mode, however, support is provided for 64-bit displacement and immediate forms of the MOV instruction.

All 16-bit and 32-bit address calculations are zero-extended in IA-32e mode to form 64-bit addresses. Address calculations are first truncated to the effective address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width. Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4 GBytes of the 64-bit mode effective addresses. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4 GBytes of the 64-bit mode effective addresses.

3.3.7.1 Canonical Addressing

In 64-bit mode, an address is considered to be in canonical form if address bits 63 through to the most-significant implemented bit by the microarchitecture are set to either all ones or all zeros.

Intel 64 architecture defines a 64-bit linear address. Implementations can support less. The first implementation of IA-32 processors with Intel 64 architecture supports a 48-bit linear address. This means a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one).

Although implementations may not use all 64 bits of the linear address, they should check bits 63 through the most-significant implemented bit to see if the address is in canonical form. If a linear-memory reference is not in canonical form, the implementation should generate an exception. In most cases, a general-protection exception (#GP) is generated. However, in the case of explicit or implied stack references, a stack fault (#SS) is generated.

Instructions that have implied stack references, by default, use the SS segment register. These include PUSH/POP-related instructions and instructions using RSP/RBP as base registers. In these cases, the canonical fault is #SS.

If an instruction uses base registers RSP/RBP and uses a segment override prefix to specify a non-SS segment, a canonical fault generates a #GP (instead of an #SS). In 64-bit mode, only FS and GS segment-overrides are applicable in this situation. Other segment override prefixes (CS, DS, ES and SS) are ignored. Note that this also means that an SS segment-override applied to a "non-stack" register reference is ignored. Such a sequence still produces a #GP for a canonical fault (and not an #SS).

3.4 BASIC PROGRAM EXECUTION REGISTERS

IA-32 architecture provides 16 basic program execution registers for use in general system and application programming (see Figure 3-4). These registers can be grouped as follows:

- **General-purpose registers.** These eight registers are available for storing operands and pointers.
- **Segment registers.** These registers hold up to six segment selectors.
- **EFLAGS (program status and control) register.** The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
- **EIP (instruction pointer) register.** The EIP register contains a 32-bit pointer to the next instruction to be executed.

3.4.1 General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

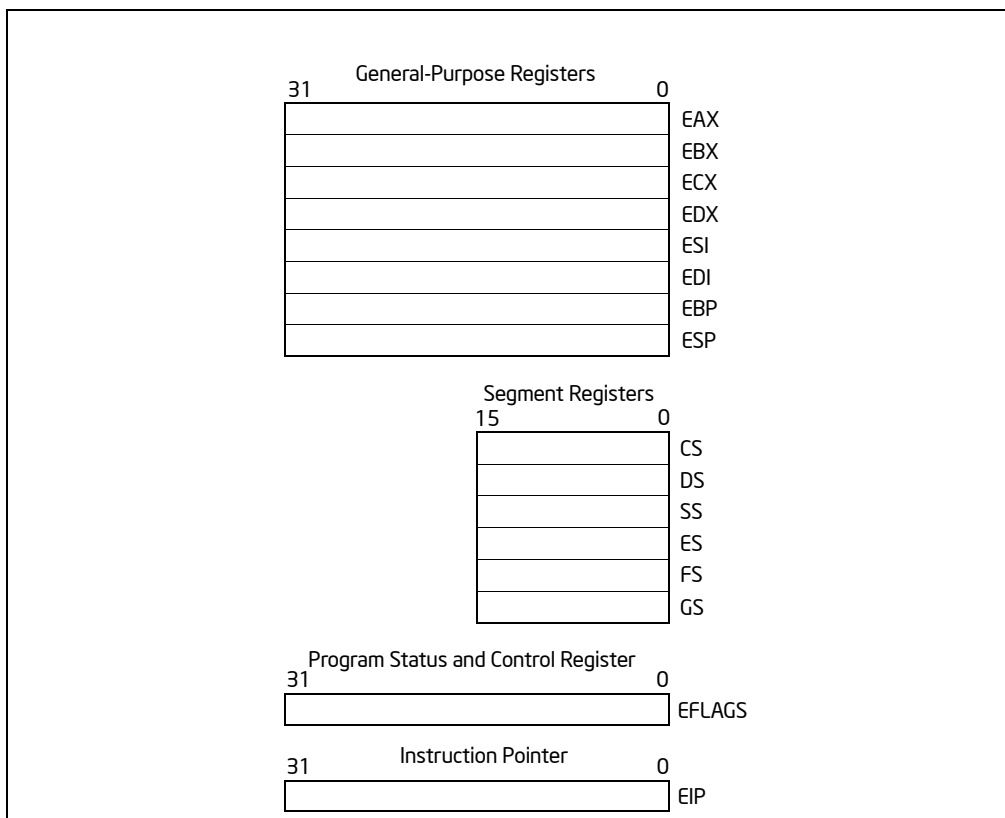


Figure 3-4. General System and Application Programming Registers

The special uses of general-purpose registers by instructions are described in Chapter 5, “Instruction Set Summary,” in this volume. See also: Chapter 3, Chapter 4 and Chapter 5 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B & 2C*. The following is a summary of special uses:

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

As shown in Figure 3-5, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

General-Purpose Registers							
31	16	15	8	7	0	16-bit	32-bit
		AH			AL	AX	EAX
		BH			BL	BX	EBX
		CH			CL	CX	ECX
		DH			DL	DX	EDX
		BP					EBP
		SI					ESI
		DI					EDI
		SP					ESP

Figure 3-5. Alternate General-Purpose Register Names

3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

Table 3-2. Addressable General Purpose Registers

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8B - R15B
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

In 64-bit mode, there are limitations on accessing byte registers. An instruction cannot reference legacy high-bytes (for example: AH, BH, CH, DH) and one of the new byte registers at the same time (for example: the low byte of the RAX register). However, instructions may reference legacy low-bytes (for example: AL, BL, CL or DL) and new byte registers at the same time (for example: the low byte of the R8 register, or RBP). The architecture enforces this limitation by changing high-byte references (AH, BH, CH, DH) to low byte references (BPL, SPL, DIL, SIL: the low 8 bits for RBP, RSP, RDI and RSI) for instructions using a REX prefix.

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Because the upper 32 bits of 64-bit general-purpose registers are undefined in 32-bit modes, the upper 32 bits of any general-purpose register are not preserved when switching from 64-bit mode to a 32-bit mode (to protected mode or compatibility mode). Software must not depend on these bits to maintain a value after a 64-bit to 32-bit mode switch.

3.4.2 Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, programmers generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If writing system code, programmers may need to create segment selectors directly. See Chapter 3, "Protected-Mode Memory Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (see Figure 3-6). These overlapping segments then comprise the linear address space for the program. Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space (see Figure 3-7). At any time, a program can thus access up to six segments in the linear address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

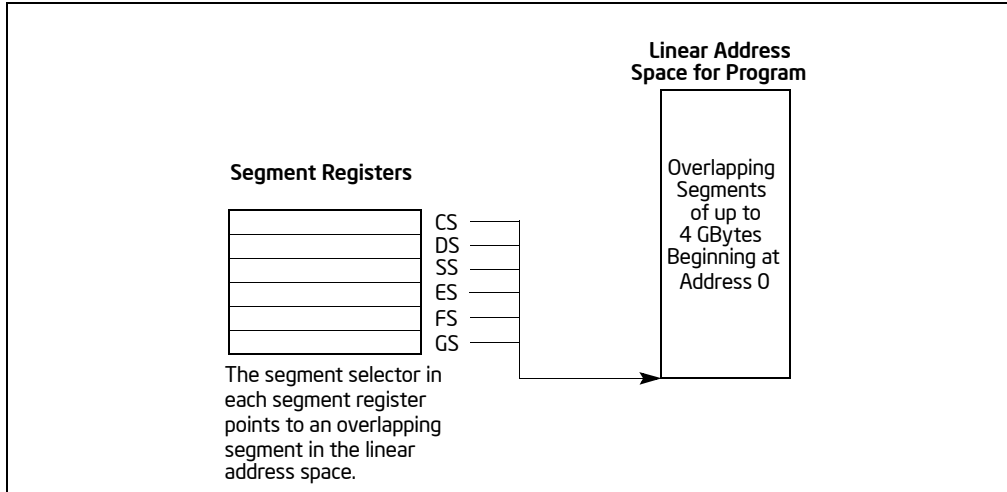


Figure 3-6. Use of Segment Registers for Flat Memory Model

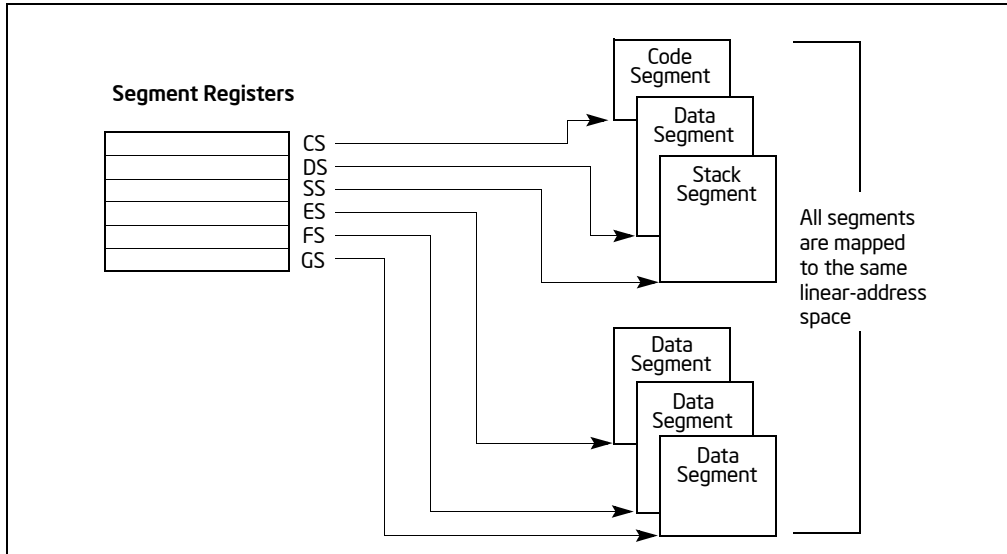


Figure 3-7. Use of Segment Registers in Segmented Memory Model

Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for the **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack

segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3, “Memory Organization,” for an overview of how the segment registers are used in real-address mode.

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the IA-32 Architecture with the Intel386™ family of processors.

3.4.2.1 Segment Registers in 64-Bit Mode

In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures).

Even though segmentation is generally disabled, segment register loads may cause the processor to perform segment access assists. During these activities, enabled processors will still perform most of the legacy checks on loaded values (even if the checks are not applicable in 64-bit mode). Such checks are needed because a segment register loaded in 64-bit mode may be used by an application running in compatibility mode.

Limit checks for CS, DS, ES, SS, FS, and GS are disabled in 64-bit mode.

3.4.3 EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-8 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly.

The following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor’s bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor’s multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task’s TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

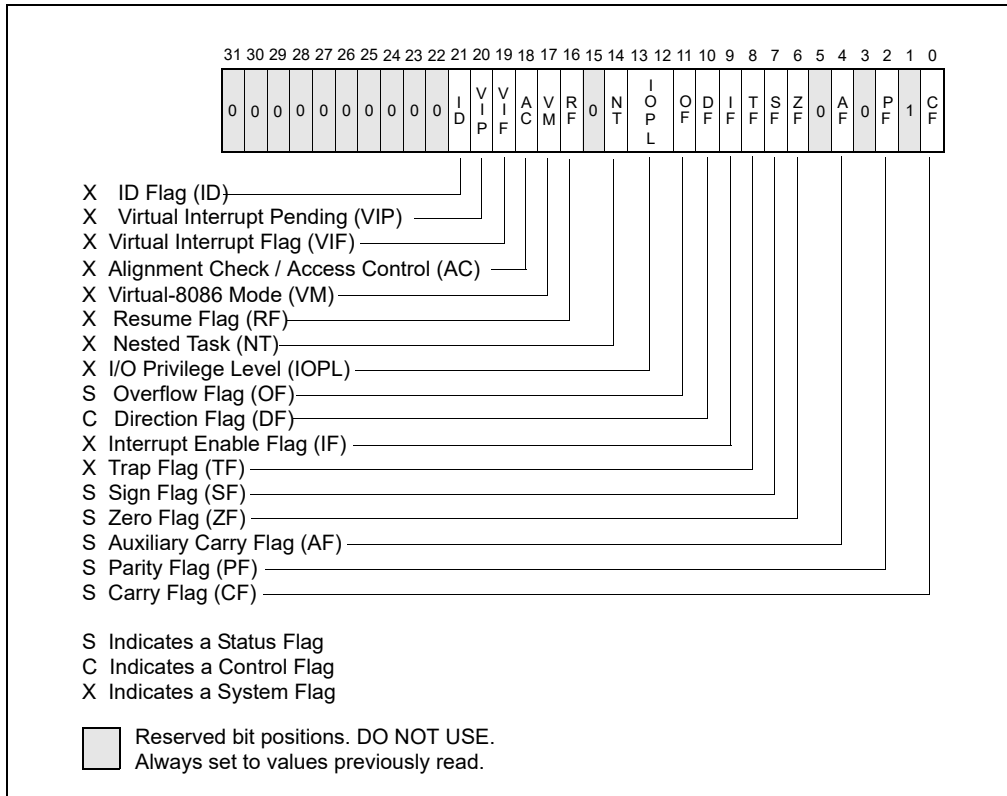


Figure 3-8. EFLAGS Register

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- CF (bit 0)** **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
- PF (bit 2)** **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
- AF (bit 4)** **Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
- ZF (bit 6)** **Zero flag** — Set if the result is zero; cleared otherwise.
- SF (bit 7)** **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
- OF (bit 11)** **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code cc), SETcc (byte set on condition code cc), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

3.4.3.2 DF Flag

The direction flag (DF, located in bit 10 of the EFLAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

3.4.3.3 System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. **They should not be modified by application programs.** The functions of the system flags are as follows:

- TF (bit 8)** **Trap flag** — Set to enable single-step mode for debugging; clear to disable single-step mode.
- IF (bit 9)** **Interrupt enable flag** — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.
- IOPL (bits 12 and 13)**
I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.
- NT (bit 14)** **Nested task flag** — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.
- RF (bit 16)** **Resume flag** — Controls the processor's response to debug exceptions.
- VM (bit 17)** **Virtual-8086 mode flag** — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.
- AC (bit 18)** **Alignment check (or access control) flag** — 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.
 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," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.
- VIF (bit 19)** **Virtual interrupt flag** — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)
- VIP (bit 20)** **Virtual interrupt pending flag** — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.
- ID (bit 21)** **Identification flag** — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

For a detailed description of these flags: see Chapter 3, “Protected-Mode Memory Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

3.4.3.4 RFLAGS Register in 64-Bit Mode

In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.

3.5 INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 6.2.4.2, “Return Instruction Pointer.”

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

3.5.1 Instruction Pointer in 64-Bit Mode

In 64-bit mode, the RIP register becomes the instruction pointer. This register holds the 64-bit offset of the next instruction to be executed. 64-bit mode also supports a technique called RIP-relative addressing. Using this technique, the effective address is determined by adding a displacement to the RIP of the next instruction.

3.6 OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When the processor is executing in protected mode, every code segment has a default operand-size attribute and address-size attribute. These attributes are selected with the D (default size) flag in the segment descriptor for the code segment (see Chapter 3, “Protected-Mode Memory Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). When the D flag is set, the 32-bit operand-size and address-size attributes are selected; when the flag is clear, the 16-bit size attributes are selected. When the processor is executing in real-address mode, virtual-8086 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-size attribute selects the size of operands. When the 16-bit operand-size attribute is in force, operands can generally be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can generally be 8 bits or 32 bits.

The address-size attribute selects the sizes of addresses used to address memory: 16 bits or 32 bits. When the 16-bit address-size attribute is in force, segment offsets and displacements are 16 bits. This restriction limits the size of a segment to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32 bits, allowing up to 4 GBytes to be addressed.

The default operand-size attribute and/or address-size attribute can be overridden for a particular instruction by adding an operand-size and/or address-size prefix to an instruction. See Chapter 2, “Instruction Format,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*. The effect of this prefix applies only to the targeted instruction.

Table 3-4 shows effective operand size and address size (when executing in protected mode or compatibility mode) depending on the settings of the D flag and the operand-size and address-size prefixes.

Table 3-3. Effective Operand- and Address-Size Attributes

D Flag in Code Segment Descriptor	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

NOTES:

Y: Yes - this instruction prefix is present.

N: No - this instruction prefix is not present.

3.6.1 Operand Size and Address Size in 64-Bit Mode

In 64-bit mode, the default address size is 64 bits and the default operand size is 32 bits. Defaults can be overridden using prefixes. Address-size and operand-size prefixes allow mixing of 32/64-bit data and 32/64-bit addresses on an instruction-by-instruction basis. Table 3-4 shows valid combinations of the 66H instruction prefix and the REX.W prefix that may be used to specify operand-size overrides in 64-bit mode. Note that 16-bit addresses are not supported in 64-bit mode.

REX prefixes consist of 4-bit fields that form 16 different values. The W-bit field in the REX prefixes is referred to as REX.W. If the REX.W field is properly set, the prefix specifies an operand size override to 64 bits. Note that software can still use the operand-size 66H prefix to toggle to a 16-bit operand size. However, setting REX.W takes precedence over the operand-size prefix (66H) when both are used.

In the case of SSE/SSE2/SSE3/SSSE3 SIMD instructions: the 66H, F2H, and F3H prefixes are mandatory for opcode extensions. In such a case, there is no interaction between a valid REX.W prefix and a 66H opcode extension prefix.

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

Table 3-4. Effective Operand- and Address-Size Attributes in 64-Bit Mode

L Flag in Code Segment Descriptor	1	1	1	1	1	1	1	1
REX.W Prefix	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	32	32	16	16	64	64	64	64
Effective Address Size	64	32	64	32	64	32	64	32

NOTES:

Y: Yes - this instruction prefix is present.

N: No - this instruction prefix is not present.

3.7 OPERAND ADDRESSING

IA-32 machine-instructions act on zero or more operands. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:

- the instruction itself (an immediate operand)
- a register
- a memory location
- an I/O port

When an instruction returns data to a destination operand, it can be returned to:

- a register
- a memory location
- an I/O port

3.7.1 Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called **immediate** operands (or simply immediates). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

3.7.2 Register Operands

Source and destination operands can be any of the following registers, depending on the instruction being executed:

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP)
- 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL)
- segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- control registers (CR0, CR2, CR3, and CR4) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the Jcc instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.

3.7.2.1 Register Operands in 64-Bit Mode

Register operands in 64-bit mode can be any of the following:

- 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, or R8-R15)
- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, or R8D-R15D)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, BP, or R8W-R15W)
- 8-bit general-purpose registers: AL, BL, CL, DL, SIL, DIL, SPL, BPL, and R8B-R15B are available using REX prefixes; AL, BL, CL, DL, AH, BH, CH, DH are available without using REX prefixes.
- Segment registers (CS, DS, SS, ES, FS, and GS)
- RFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM15) and the MXCSR register
- Control registers (CR0, CR2, CR3, CR4, and CR8) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers
- RDX:RAX register pair representing a 128-bit operand

3.7.3 Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 3-9). Segment selectors specify the segment containing the operand. Offsets specify the linear or effective address of the operand. Offsets can be 32 bits (represented by the notation m16:32) or 16 bits (represented by the notation m16:16).

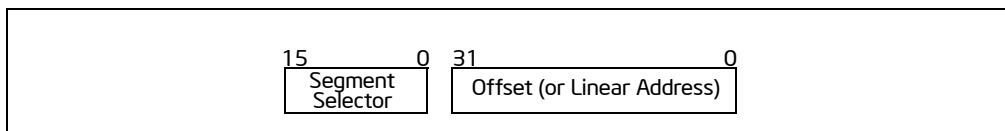


Figure 3-9. Memory Operand Address

3.7.3.1 Memory Operands in 64-Bit Mode

In 64-bit mode, a memory operand can be referenced by a segment selector and an offset. The offset can be 16 bits, 32 bits or 64 bits (see Figure 3-10).

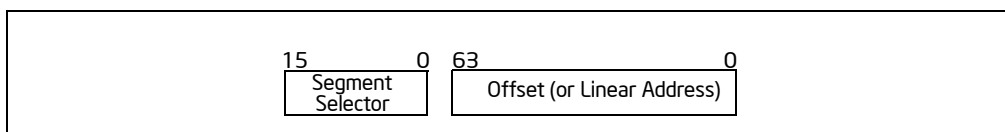


Figure 3-10. Memory Operand Address in 64-Bit Mode

3.7.4 Specifying a Segment Selector

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the rules given in Table 3-5.

When storing data in memory or loading data from memory, the DS segment default can be overridden to allow other segments to be accessed. Within an assembler, the segment override is generally handled with a colon ":" operator. For example, the following MOV instruction moves a value from register EAX into the segment pointed to by the ES register. The offset into the segment is contained in the EBX register:

```
MOV ES:[EBX], EAX
```

Table 3-5. Default Segment Selection Rules

Reference Type	Register Used	Segment Used	Default Selection Rule
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

At the machine level, a segment override is specified with a segment-override prefix, which is a byte placed at the beginning of an instruction. The following default segment selections cannot be overridden:

- Instruction fetches must be made from the code segment.
- Destination strings in string instructions must be stored in the data segment pointed to by the ES register.
- Push and pop operations must always reference the SS segment.

Some instructions require a segment selector to be specified explicitly. In these cases, the 16-bit segment selector can be located in a memory location or in a 16-bit register. For example, the following MOV instruction moves a segment selector located in register BX into segment register DS:

```
MOV DS, BX
```

Segment selectors can also be specified explicitly as part of a 48-bit far pointer in memory. Here, the first double-word in memory contains the offset and the next word contains the segment selector.

3.7.4.1 Segmentation in 64-Bit Mode

In IA-32e mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does in legacy IA-32 mode, using the 16-bit or 32-bit protected mode semantics described above.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as additional base registers in some linear address calculations.

3.7.5 Specifying an Offset

The offset part of a memory address can be specified directly as a static value (called a **displacement**) or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-, 16-, or 32-bit value.
- **Base** — The value in a general-purpose register.
- **Index** — The value in a general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2's complement) value, with the exception of the scaling factor. Figure 3-11 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

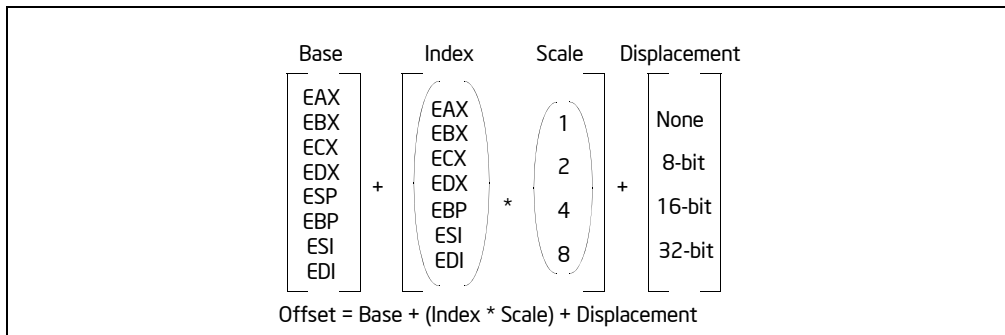


Figure 3-11. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be NULL. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language.

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
 - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
 - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

- **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.
- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index * Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

3.7.5.1 Specifying an Offset in 64-Bit Mode

The offset part of a memory address in 64-bit mode can be specified directly as a static value or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-bit, 16-bit, or 32-bit value.
- **Base** — The value in a 64-bit general-purpose register.
- **Index** — The value in a 64-bit general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The base and index value can be specified in one of sixteen available general-purpose registers in most cases. See Chapter 2, "Instruction Format," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The following unique combination of address components is also available.

- **RIP + Displacement** — In 64-bit mode, RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP.

3.7.6 Assembler and Compiler Addressing Modes

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language compilers will select an appropriate combination of these components based on the language construct a programmer defines.

3.7.7 I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 19, "Input/Output," for more information about I/O port addressing.

3. Updates to Chapter 5, Volume 1

Updates to Chapter 5 added to the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

Changes to this chapter: Update to the CLWB entry in Table 5-2 "Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors".

CHAPTER 5

INSTRUCTION SET SUMMARY

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

- Section 5.1, “General-Purpose Instructions”.
- Section 5.2, “x87 FPU Instructions”.
- Section 5.3, “x87 FPU AND SIMD State Management Instructions”.
- Section 5.4, “MMX™ Instructions”.
- Section 5.5, “SSE Instructions”.
- Section 5.6, “SSE2 Instructions”.
- Section 5.7, “SSE3 Instructions”.
- Section 5.8, “Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions”.
- Section 5.9, “SSE4 Instructions”.
- Section 5.10, “SSE4.1 Instructions”.
- Section 5.11, “SSE4.2 Instruction Set”.
- Section 5.12, “Intel® AES-NI and PCLMULQDQ”.
- Section 5.13, “Intel® Advanced Vector Extensions (Intel® AVX)”.
- Section 5.14, “16-bit Floating-Point Conversion”.
- Section 5.15, “Fused-Multiply-ADD (FMA)”.
- Section 5.16, “Intel® Advanced Vector Extensions 2 (Intel® AVX2)”.
- Section 5.17, “Intel® Transactional Synchronization Extensions (Intel® TSX)”.
- Section 5.18, “Intel® SHA Extensions”.
- Section 5.19, “Intel® Advanced Vector Extensions 512 (Intel® AVX-512)”.
- Section 5.20, “System Instructions”.
- Section 5.21, “64-Bit Mode Instructions”.
- Section 5.22, “Virtual-Machine Extensions”.
- Section 5.23, “Safer Mode Extensions”.
- Section 5.24, “Intel® Memory Protection Extensions”.
- Section 5.25, “Intel® Software Guard Extensions”.
- Section 5.26, “Shadow Stack Management Instructions”.
- Section 5.27, “Control Transfer Terminating Instructions”.

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

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

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

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

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

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

Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel® Core™ 2 Extreme processors QX9000 series, Intel® Core™ 2 Quad processor Q9000 series, Intel® Core™ 2 Duo processors 8000 series and T9000 series, Intel Atom® processor based on Silvermont microarchitecture.
SSE4.2 Extensions, CRC32, POPCNT	Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series, Intel Atom processor based on Silvermont microarchitecture.
Intel® AES-NI, PCLMULQDQ	Intel® Xeon® processor E7 series, Intel® Xeon® processors X3600 and X5600, Intel® Core™ i7 980X processor, Intel Atom processor based on Silvermont microarchitecture. Use CPUID to verify presence of Intel AES-NI and PCLMULQDQ across Intel® Core™ processor families.
Intel® AVX	Intel® Xeon® processor E3 and E5 families, 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.
F16C	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Intel® Xeon® processor E5 v2 and E7 v2 families.
RDRAND	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom processor based on Silvermont microarchitecture.
FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom® processor based on Goldmont microarchitecture.
FMA, AVX2, BMI1, BMI2, INVPCID, LZCNT, Intel® TSX	Intel® Xeon® processor E3/E5/E7 v3 product families, 4th Generation Intel® Core™ processor family.
MOVBE	Intel Xeon processor E3/E5/E7 v3 product families, 4th Generation Intel Core processor family, Intel Atom processors.
PREFETCHW	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family, Intel Atom processor based on Silvermont microarchitecture.

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

Instruction Set Architecture	Processor Generation Introduction
Intel® SHA Extensions	Intel Atom processor based on Goldmont microarchitecture.
ADX	Intel Core M processor family, 5th Generation Intel Core processor family.
RDSEED, CLAC, STAC	Intel Core M processor family, 5th Generation Intel Core processor family, Intel Atom processor based on Goldmont microarchitecture.
AVX512ER, AVX512PF, PREFETCHWT1	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.
AVX512F, AVX512CD	Intel Xeon Phi Processor 3200, 5200, 7200 Series, Intel® Xeon® Processor Scalable Family, Intel® Core™ i3-8121U processor.
CLFLUSHOPT, XSAVEC, XSAVES, Intel® MPX	Intel Xeon Processor Scalable Family, 6th Generation Intel® Core™ processor family, Intel Atom processor based on Goldmont microarchitecture.
SGX1	6th Generation Intel Core processor family, Intel Atom® processor based on Goldmont Plus microarchitecture.
AVX512DQ, AVX512BW, AVX512VL	Intel Xeon Processor Scalable Family, Intel Core i3-8121U processor.
CLWB	Intel Xeon Processor Scalable Family, Intel Atom® processor based on Tremont microarchitecture.
PKU	Intel Xeon Processor Scalable Family.
AVX512_IFMA, AVX512_VBMI	Intel Core i3-8121U processor.
SHA-NI	Intel Core i3-8121U processor, Intel Atom processor based on Goldmont microarchitecture.
UMIP	Intel Core i3-8121U processor, Intel Atom processor based on Goldmont Plus microarchitecture.
PTWRITE	Intel Atom processor based on Goldmont Plus microarchitecture.
RDPID	10th Generation Intel® Core™ processor family, Intel Atom processor based on Goldmont Plus microarchitecture.
AVX512_4FMAPS, AVX512_4VNNIW	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.
AVX512_VNNI	2nd Generation Intel® Xeon® Processor Scalable Family, 10th Generation Intel Core processor family.
AVX512_VPOPCNTDQ	Intel Xeon Phi Processor 7215, 7285, 7295 Series, 10th Generation Intel Core processor family.
Fast Short REP MOV	10th Generation Intel Core processor family.
GFNI (SSE)	10th Generation Intel Core processor family, Intel Atom processor based on Tremont microarchitecture.
VAES, GFNI (AVX/AVX512), AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG	10th Generation Intel Core processor family.
ENCLV	Intel Atom processor based on Tremont microarchitecture.
Split Lock Detection	10th Generation Intel Core processor family, Intel Atom processor based on Tremont microarchitecture.
CLDEMOTTE	Intel Atom processor based on Tremont microarchitecture.
Direct stores: MOVDIRI, MOVDIR64B	Intel Atom processor based on Tremont microarchitecture.
User wait: TPAUSE, UMONITOR, UMWAIT	Intel Atom processor based on Tremont microarchitecture.

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

Instruction Set Architecture	Processor Generation Introduction
Control-flow Enforcement Technology (CET)	Future Intel Core processors.

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

5.1 GENERAL-PURPOSE INSTRUCTIONS

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

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

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

5.1.1 Data Transfer Instructions

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

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVBE/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).

CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.
MOVZX	Move and zero extend.

5.1.2 Binary Arithmetic Instructions

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

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

5.1.3 Decimal Arithmetic Instructions

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

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

5.1.4 Logical Instructions

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

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

5.1.5 Shift and Rotate Instructions

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

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.
SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

5.1.6 Bit and Byte Instructions

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

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.
BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.

TEST	Logical compare.
CRC32 ¹	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT ²	This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

5.1.7 Control Transfer Instructions

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

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

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

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

5.1.8 String Instructions

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

MOVS/MOVS	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.
STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPNZ	Repeat while not equal/Repeat while not zero.

5.1.9 I/O Instructions

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

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

5.1.10 Enter and Leave Instructions

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

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

5.1.11 Flag Control (EFLAG) Instructions

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

STC	Set carry flag.
CLC	Clear the carry flag.
CMC	Complement the carry flag.
CLD	Clear the direction flag.

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

5.1.12 Segment Register Instructions

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

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

5.1.13 Miscellaneous Instructions

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

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

5.1.14 User Mode Extended State Save/Restore Instructions

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

5.1.15 Random Number Generator Instructions

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

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

5.1.16 BMI1, BMI2

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

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

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

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

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

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

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

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.

5.2 X87 FPU INSTRUCTIONS

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

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

5.2.1 x87 FPU Data Transfer Instructions

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

FLD	Load floating-point value.
FST	Store floating-point value.
FSTP	Store floating-point value and pop.
FILD	Load integer.
FIST	Store integer.

FISTP ¹	Store integer and pop.
FBLD	Load BCD.
FBSTP	Store BCD and pop.
FXCH	Exchange registers.
FCMOVE	Floating-point conditional move if equal.
FCMOVNE	Floating-point conditional move if not equal.
FCMOVB	Floating-point conditional move if below.
FCMOVBE	Floating-point conditional move if below or equal.
FCMOVNB	Floating-point conditional move if not below.
FCMOVNBE	Floating-point conditional move if not below or equal.
FCMOVU	Floating-point conditional move if unordered.
FCMOVNU	Floating-point conditional move if not unordered.

5.2.2 x87 FPU Basic Arithmetic Instructions

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

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

1. SSE3 provides an instruction FISTTP for integer conversion.

5.2.3 x87 FPU Comparison Instructions

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

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

5.2.4 x87 FPU Transcendental Instructions

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

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

5.2.5 x87 FPU Load Constants Instructions

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

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

5.2.6 x87 FPU Control Instructions

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

FINCSTP	Increment FPU register stack pointer.
FDECSTP	Decrement FPU register stack pointer.
FFREE	Free floating-point register.
FINIT	Initialize FPU after checking error conditions.

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

5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

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

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

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

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

5.4 MMX™ INSTRUCTIONS

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

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

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

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

5.4.1 MMX Data Transfer Instructions

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

MOVD	Move doubleword.
MOVQ	Move quadword.

5.4.2 MMX Conversion Instructions

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

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

5.4.3 MMX Packed Arithmetic Instructions

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

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

5.4.4 MMX Comparison Instructions

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

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

PCMPGTW	Compare packed signed word integers for greater than.
PCMPGTD	Compare packed signed doubleword integers for greater than.

5.4.5 MMX Logical Instructions

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

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

5.4.6 MMX Shift and Rotate Instructions

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

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

5.4.7 MMX State Management Instructions

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

EMMS	Empty MMX state.
------	------------------

5.5 SSE INSTRUCTIONS

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

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

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

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

The following sections provide an overview of these groups.

5.5.1 SSE SIMD Single-Precision Floating-Point Instructions

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

5.5.1.1 SSE Data Transfer Instructions

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

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

5.5.1.2 SSE Packed Arithmetic Instructions

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

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

5.5.1.3 SSE Comparison Instructions

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

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

5.5.1.4 SSE Logical Instructions

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

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

5.5.1.5 SSE Shuffle and Unpack Instructions

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

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

5.5.1.6 SSE Conversion Instructions

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

CVTPI2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTSI2SS	Convert doubleword integer to scalar single-precision floating-point value.
CVTSS2PI	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTSS2PI	Convert with truncation packed single-precision floating-point values to packed doubleword integers.
CVTSS2SI	Convert a scalar single-precision floating-point value to a doubleword integer.
CVTTSS2SI	Convert with truncation a scalar single-precision floating-point value to a scalar doubleword integer.

5.5.2 SSE MXCSR State Management Instructions

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

LDMXCSR	Load MXCSR register.
STMXCSR	Save MXCSR register state.

5.5.3 SSE 64-Bit SIMD Integer Instructions

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

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

5.5.4 SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

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

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

5.6 SSE2 INSTRUCTIONS

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

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

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

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

The following sections give an overview of each subgroup.

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

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

5.6.1.1 SSE2 Data Movement Instructions

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

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

5.6.1.2 SSE2 Packed Arithmetic Instructions

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

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

5.6.1.3 SSE2 Logical Instructions

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

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

5.6.1.4 SSE2 Compare Instructions

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

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

5.6.1.5 SSE2 Shuffle and Unpack Instructions

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

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

5.6.1.6 SSE2 Conversion Instructions

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

CVTPD2PI	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2PI	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTPI2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTPD2DQ	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2DQ	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTDQ2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTSS2PD	Convert packed single-precision floating-point values to packed double-precision floating-point values.
CVTPD2PS	Convert packed double-precision floating-point values to packed single-precision floating-point values.
CVTSS2SD	Convert scalar single-precision floating-point values to scalar double-precision floating-point values.
CVTSD2SS	Convert scalar double-precision floating-point values to scalar single-precision floating-point values.
CVTSD2SI	Convert scalar double-precision floating-point values to a doubleword integer.
CVTTSD2SI	Convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
CVTSI2SD	Convert doubleword integer to scalar double-precision floating-point value.

5.6.2 SSE2 Packed Single-Precision Floating-Point Instructions

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

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

5.6.3 SSE2 128-Bit SIMD Integer Instructions

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

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword.
MOVQ2DQ	Move quadword integer from MMX to XMM registers.
MOVDQ2Q	Move quadword integer from XMM to MMX registers.
PMULUDQ	Multiply packed unsigned doubleword integers.
PADDQ	Add packed quadword integers.
PSUBQ	Subtract packed quadword integers.
PSHUFLW	Shuffle packed low words.
PSHUFHW	Shuffle packed high words.
PSHUFDB	Shuffle packed doublewords.
PSLLDQ	Shift double quadword left logical.
PSRLDQ	Shift double quadword right logical.
PUNPCKHQDQ	Unpack high quadwords.
PUNPCKLQDQ	Unpack low quadwords.

5.6.4 SSE2 Cacheability Control and Ordering Instructions

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

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

5.7 SSE3 INSTRUCTIONS

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

- One x87FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.

- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

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

The sections that follow describe each subgroup.

5.7.1 SSE3 x87-FP Integer Conversion Instruction

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

5.7.2 SSE3 Specialized 128-bit Unaligned Data Load Instruction

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

5.7.3 SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

ADDSSUBPS Performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; single-precision subtraction on the first and third pairs.

ADDSSUBPD Performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.

5.7.4 SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

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

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

HADDPD Performs a double-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.

HSUBPD Performs a double-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

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

MOVSHDUP Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.

MOVSLDUP Loads/moves 128 bits; duplicating the first and third 32-bit data elements.

MOVDDUP Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

5.7.6 SSE3 Agent Synchronization Instructions

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

5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

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

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

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

The sections that follow describe each subgroup.

5.8.1 Horizontal Addition/Subtraction

PHADDW Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.

PHADDSW Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.

PHADDD Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.

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. The signed 16-bit results are packed and written to the destination operand.

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 and written to the destination operand.

PHSUBD Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

5.8.2 Packed Absolute Values

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

5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

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

5.8.4 Packed Multiply High with Round and Scale

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

5.8.5 Packed Shuffle Bytes

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

5.8.6 Packed Sign

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

5.8.7 Packed Align Right

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

5.9 SSE4 INSTRUCTIONS

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

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

The 47 SSE4.1 instructions include:

- Two instructions perform packed dword multiplies.

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

The SSE4.2 instructions operating on XMM registers include:

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

5.10 SSE4.1 INSTRUCTIONS

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

5.10.1 Dword Multiply Instructions

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

5.10.2 Floating-Point Dot Product Instructions

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

5.10.3 Streaming Load Hint Instruction

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

5.10.4 Packed Blending Instructions

BLENDPD	Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
---------	--

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

5.10.5 Packed Integer MIN/MAX Instructions

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

5.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

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

5.10.7 Insertion and Extractions from XMM Registers

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

PEXTRQ Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

5.10.8 Packed Integer Format Conversions

PMOVSXBW	Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVZXBW	Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVSXBD	Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVZXBW	Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVSXWD	Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVZXWD	Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers..
PMOVSXBQ	Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVZXBQ	Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVSXWQ	Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVZXWQ	Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVSXDQ	Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.
PMOVZXDQ	Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

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

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

5.10.10 Horizontal Search

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

5.10.11 Packed Test

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

5.10.12 Packed Qword Equality Comparisons

PCMPEQQ 128-bit packed qword equality test.

5.10.13 Dword Packing With Unsigned Saturation

PACKUSDW **PACKUSDW** packs dword to word with unsigned saturation.

5.11 SSE4.2 INSTRUCTION SET

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

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

5.11.1 String and Text Processing Instructions

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

5.11.2 Packed Comparison SIMD integer Instruction

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

5.12 INTEL® AES-NI AND PCLMULQDQ

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

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

5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

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

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

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

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

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.

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

5.14 16-BIT FLOATING-POINT CONVERSION

Conversion between single-precision floating-point (32-bit) and half-precision FP (16-bit) data are provided by VCVT2PS2PH, VCVT2PH2PS:

VCVT2PH2PS	Convert eight/four data element containing 16-bit floating-point data into eight/four single-precision floating-point data.
VCVT2PS2PH	Convert eight/four data element containing single-precision floating-point data into eight/four 16-bit floating-point data.

5.15 FUSED-MULTIPLY-ADD (FMA)

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

- Table 14-15 lists FMA instruction sets.

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

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

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

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

5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

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

5.18 INTEL® SHA EXTENSIONS

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

SHA1MSG1	Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
SHA1MSG2	Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
SHA1NEXTE	Calculate SHA1 state E after four rounds.
SHA1RND4	Perform four rounds of SHA1 operations.
SHA256MSG1	Perform an intermediate calculation for the next four SHA256 message dwords.
SHA256MSG2	Perform the final calculation for the next four SHA256 message dwords.
SHA256RND2	Perform two rounds of SHA256 operations.

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

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

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX / Intel AVX2 but with enhancement provided by opmask registers not available to VEX-encoded Intel AVX / Intel AVX2. Some instruction mnemonics in AVX / AVX2 that are promoted into AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBROADCASTF128 into VBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.6, “Intel® AVX-512 Encoding” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

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

VALIGND/Q	Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS	Replace the VBLENDVPD/PS instructions (using opmask as select control).
VCOMPRESSPD/PS	Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ	Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ	Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQQ2PD/PS	Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI	Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI	Convert the low SP FP element of a vector to an unsigned integer.
VCVTUDQ2PD/PS	Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S	Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANPD/PS	Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VEXTRACTI32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIMMPD/PS	Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIMMSD/SS	Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS	Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS	Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS	Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS	Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSERTF32X4/64X4	Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVDQA32/64	VMOVDQA with 32/64-bit granular conditional update.
VMOVDQU32/64	VMOVDQU with 32/64-bit granular conditional update.

VPBLENDMD/Q	Blend dword/qword elements using opmask as select control.
VPBROADCAST/Q	Broadcast from general-purpose register to vector register.
VPCMPD/UD	Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ	Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D	Compress packed 64/32-bit elements of a vector.
VPERMI2D/Q	Full permute of two tables of dword/qword elements overwriting the index vector.
VPERMI2PD/PS	Full permute of two tables of DP/SP elements overwriting the index vector.
VPERMT2D/Q	Full permute of two tables of dword/qword elements overwriting one source table.
VPERMT2PD/PS	Full permute of two tables of DP/SP elements overwriting one source table.
VPEXPANDD/Q	Expand packed dword/qword elements of a vector.
VPMAXSQ	Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ	Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ	Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ	Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S US)QB	Down convert qword elements in a vector to byte elements using truncation (saturation unsigned saturation).
VPMOV(S US)QW	Down convert qword elements in a vector to word elements using truncation (saturation unsigned saturation).
VPMOV(S US)QD	Down convert qword elements in a vector to dword elements using truncation (saturation unsigned saturation).
VPMOV(S US)DB	Down convert dword elements in a vector to byte elements using truncation (saturation unsigned saturation).
VPMOV(S US)DW	Down convert dword elements in a vector to word elements using truncation (saturation unsigned saturation).
VPROLD/Q	Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q	Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q	Rotate dword/qword element right by a constant shift count with conditional update.
VPRORRD/Q	Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ	Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ	Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ	Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAVQ	Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTSTNMD/Q	Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTERLOGD/Q	Perform bitwise ternary logic operation of three vectors with 32/64 bit granular conditional update.
VPTSTMD/Q	Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS	Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS	Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS	Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS	Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS	Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS	Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCALEPD/PS	Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in a second vector.

VSCALESD/SS	Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ	Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ	Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFI32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

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

VCVT(T)PD2QQ	Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ	Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ	Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ	Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS	Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2	Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2	Extract a vector from a full-length vector with 64-bit granular update.
VFPCLASSPD/PS	Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFPCLASSSD/SS	Test the low DP/SP FP element by numeric/special-value category.
VINSERTF64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSERTI64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VPMOVM2D/Q	Convert opmask register to vector register in 32/64-bit granularity.
VPMOVB2D/Q2M	Convert a vector register in 32/64-bit granularity to an opmask register.
VPMULLQ	Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.
VRANGEPD/PS	Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.
VRANGESD/SS	Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.
VREDUCEPD/PS	Perform Reduction operation on packed DP/SP FP elements of a vector using specified reduction primitive in imm8.
VREDUCESD/SS	Perform Reduction operation on the low DP/SP FP element of a vector using specified reduction primitive in imm8.

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

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

VPSRLVW	Shift word elements in a vector right by shift counts in a vector.
VPTESTNMB/W	Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.
VPTESTMB/W	Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

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

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

Opmask instructions include:

KADDB/W/D/Q	Add two 8/16/32/64-bit opmasks.
KANDB/W/D/Q	Logical AND two 8/16/32/64-bit opmasks.
KANDNB/W/D/Q	Logical AND NOT two 8/16/32/64-bit opmasks.
KMOVB/W/D/Q	Move from or move to opmask register of 8/16/32/64-bit data.
KNOTB/W/D/Q	Bitwise NOT of two 8/16/32/64-bit opmasks.
KORB/W/D/Q	Logical OR two 8/16/32/64-bit opmasks.
KORTESTB/W/D/Q	Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.
KSHIFTLB/W/D/Q	Shift left 8/16/32/64-bit opmask by specified count.
KSHIFTRB/W/D/Q	Shift right 8/16/32/64-bit opmask by specified count.
KTESTB/W/D/Q	Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ	Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q	Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
KXORB/W/D/Q	Logical XOR of two 8/16/32/64-bit opmasks.

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

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

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

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

5.20 SYSTEM INSTRUCTIONS

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

CLAC	Clear AC Flag in EFLAGS register.
STAC	Set AC Flag in EFLAGS register.
LGDT	Load global descriptor table (GDT) register.
SGDT	Store global descriptor table (GDT) register.
LLDT	Load local descriptor table (LDT) register.
SLDT	Store local descriptor table (LDT) register.
LTR	Load task register.
STR	Store task register.
LIDT	Load interrupt descriptor table (IDT) register.
SIDT	Store interrupt descriptor table (IDT) register.
MOV	Load and store control registers.
LMSW	Load machine status word.
SMSW	Store machine status word.
CLTS	Clear the task-switched flag.
ARPL	Adjust requested privilege level.
LAR	Load access rights.
LSL	Load segment limit.
VERR	Verify segment for reading
VERW	Verify segment for writing.
MOV	Load and store debug registers.
INVD	Invalidate cache, no writeback.
WBINVD	Invalidate cache, with writeback.
INVLPG	Invalidate TLB Entry.
INVPCID	Invalidate Process-Context Identifier.
LOCK (prefix)	Perform atomic access to memory (can be applied to a number of general purpose instructions that provide memory source/destination access).
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMC	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XSAVES	Save processor supervisor-mode extended states to memory.
XRSTOR	Restore processor extended states from memory.
XRSTORS	Restore processor supervisor-mode extended states from memory.
XGETBV	Reads the state of an extended control register.
XSETBV	Writes the state of an extended control register.

RDFSBASE	Reads from FS base address at any privilege level.
RDGSBASE	Reads from GS base address at any privilege level.
WRFSBASE	Writes to FS base address at any privilege level.
WRGSBASE	Writes to GS base address at any privilege level.

5.21 64-BIT MODE INSTRUCTIONS

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

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

5.22 VIRTUAL-MACHINE EXTENSIONS

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

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

The behavior of the VMX management instructions is summarized below:

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

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

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

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

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

VMCALL	Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC	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. No VM exit occurs.

5.23 SAFER MODE EXTENSIONS

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

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

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

GETSEC[EXITAC] Exits authenticated code execution mode.

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

GETSEC[SEXIT] Exits the MLE.

GETSEC[PARAMETERS] Returns SMX related parameter information.

GETSEC[SMCTRL] SMX mode control.

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

5.24 INTEL® MEMORY PROTECTION EXTENSIONS

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

BNDMK Create a LowerBound and a UpperBound in a register.

BNDCL Check the address of a memory reference against a LowerBound.

BNDUCU Check the address of a memory reference against an UpperBound in 1's compliment form.

BNDCN Check the address of a memory reference against an UpperBound not in 1's compliment form.

BNDMOV Copy or load from memory of the LowerBound and UpperBound to a register.

BNDMOV Store to memory of the LowerBound and UpperBound from a register.

BNDLDX Load bounds using address translation.

BNDSTX Store bounds using address translation.

5.25 INTEL® SOFTWARE GUARD EXTENSIONS

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

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

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

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

5.26 SHADOW STACK MANAGEMENT INSTRUCTIONS

Shadow stack management instructions allow the program and run-time to perform operations like recovering from control protection faults, shadow stack switching, etc. The following instructions are provided.

CLRSSBSY	Clear busy bit in a supervisor shadow stack token.
INCSSP	Increment the shadow stack pointer (SSP).
RDSSP	Read shadow stack point (SSP).
RSTORSSP	Restore a shadow stack pointer (SSP).
SAVEPREVSSP	Save previous shadow stack pointer (SSP).
SETSSBSY	Set busy bit in a supervisor shadow stack token.
WRSS	Write to a shadow stack.
WRUSS	Write to a user mode shadow stack.

5.27 CONTROL TRANSFER TERMINATING INSTRUCTIONS

ENDBR32	Terminate an Indirect Branch in 32-bit and Compatibility Mode.
ENDBR64	Terminate an Indirect Branch in 64-bit Mode.

4. Updates to Chapter 13, Volume 1

Updates to Chapter 13 added to the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

Changes to this chapter: Updates related to hardware P-states (HWP state). Other minor updates and typo corrections throughout the chapter.

CHAPTER 13

MANAGING STATE USING THE XSAVE FEATURE SET

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

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

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

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

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

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

13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

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

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

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

- State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0_H–ZMM15_H (**ZMM_Hi256 state**).
- State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (**Hi16_ZMM state**).
- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (**PT state**).
- Bit 9 corresponds to the state component used for the protection-key feature’s register PKRU (**PKRU state**). See Section 13.5.7.
- Bits 12:11 correspond to the two state components used for the additional register state used by Control-Flow Enforcement Technology (**CET state**):
 - State component 11 is used for the 2 MSRs controlling user-mode functionality for CET (**CET_U state**).
 - State component 12 is used for the 3 MSRs containing shadow-stack pointers for privilege levels 0–2 (**CET_S state**).
- Bit 13 corresponds to the state component used for an MSR used to control hardware duty cycling (**HDC state**). See Section 13.5.9.
- Bit 16 corresponds to the state component used for an MSR used to control hardware P-states (**HWP state**). See Section 13.5.10.

Bit 10, bits 15:14, and bits in the range 62:17 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state components are defined using those bits, additional sub-sections will be updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

The state component corresponding to bit *i* of state-component bitmaps is called **state component *i***. Thus, x87 state is state component 0; SSE state is state component 1; AVX state is state component 2; MPX state comprises state components 3–4; AVX-512 state comprises state components 5–7; PT state is state component 8; PKRU state is state component 9; CET state comprises state components 11–12; HDC state is state component 13; and HWP state is state component 16.

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

Some state components are **user state components**, and they can be managed by the entire XSAVE feature set. Other state components are **supervisor state components**, and they can be managed only by XSAVES and XRSTORS. The state components corresponding to bit 9 and to bits in the range 7:0 are user state components; those corresponding to bit 8, to bits in the range 13:11, and to bit 16 are supervisor state components.

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

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

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

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

The state components for x87 state, for SSE state, for PT state, for PKRU state, for CET state, for HDC state, and for HWP state are XSAVE-managed but the corresponding features are not XSAVE-enabled. Processors allow modi-

ification of this state, as well as execution of x87 FPU instructions and SSE instructions and use of Intel Processor Trace, protection keys, CET, hardware duty cycling, and hardware P-states regardless of the value of CR4.OSXSAVE and XCR0.

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

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

- CPUID.1:ECX.XSAVE[bit 26] enumerates general support for the XSAVE feature set:
 - If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).
 - If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV.¹ Further enumeration is provided through CPUID function 0DH.
- CR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.
- CPUID function 0DH enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:
 - CPUID function 0DH, sub-function 0.
 - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).
If $EAX[i] = 1$ (for $1 < i < 32$) or $EDX[i-32] = 1$ (for $32 \leq i < 63$), sub-function i enumerates details for state component i (see below).
 - ECX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.
 - EBX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.
 - CPUID function 0DH, sub-function 1.
 - EAX[0] enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If $EAX[0] = 0$, execution of XSAVEOPT causes an invalid-opcode exception (#UD).
 - EAX[1] enumerates support for **compaction extensions** to the XSAVE feature set. The following are supported if this bit is 1:
 - The compacted format of the extended region of XSAVE areas (see Section 13.4.3).
 - The XSAVEC instruction. If $EAX[1] = 0$, execution of XSAVEC causes a #UD.
 - Execution of the compacted form of XRSTOR (see Section 13.8).
 - EAX[2] enumerates support for execution of XGETBV with $ECX = 1$. This allows software to determine the state of the init optimization. See Section 13.6.
 - EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32_XSS MSR. If $EAX[3] = 0$, execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32_XSS) will also set EAX[1] (the compaction extensions).
 - EAX[31:4] are reserved.

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

- EBX enumerates the size (in bytes) required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32_XSS.
- EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32_XSS MSR if and only if the corresponding bit is set in this bitmap.

NOTE

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

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

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

13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES

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

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

- XCR0[0] is associated with x87 state (see Section 13.5.1). XCR0[0] is always 1. It has that value coming out of RESET. Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[0] is 0.
- XCR0[1] is associated with SSE state (see Section 13.5.2). Software can use the XSAVE feature set to manage SSE state only if XCR0[1] = 1. The value of XCR0[1] in no way determines whether software can execute SSE instructions (these instructions can be executed even if XCR0[1] = 0).
XCR0[1] is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set XCR0[1].
- XCR0[2] is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if XCR0[2] = 1. In addition, software can execute AVX instructions only if CR4.OSXSAVE = XCR0[2] = 1. Otherwise, any execution of an AVX instruction causes an invalid-opcode exception (#UD).

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

As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears XCR0[2]. However, clearing XCR0[2] while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 13.5.3, “Enable the Use Of XSAVE Feature Set And XSAVE State Components” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[4:3] are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if $\text{XCR0}[4:3] = 11\text{b}$. In addition, MPX instructions **operate as defined** only if $\text{CR4.OSXSAVE} = 1$ and $\text{XCR0}[4:3] = 11\text{b}$. Otherwise, **execution of an MPX instruction causes no operation (as a NOP instruction); in addition, executions of CALL, RET, JMP, and Jcc do not initialize the bounds registers, and they ignore any F2H (BND) prefix.**¹

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

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

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

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

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

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

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

- XCR0[63:10] and XCR0[8] are reserved.² Executing the XSETBV instruction causes a general-protection fault (#GP) if $\text{ECX} = 0$ and any corresponding bit in $\text{EDX}:\text{EAX}$ is not 0. These bits in XCR0 are all 0 coming out of RESET.

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

1. **Prior to the introduction of MPX, the opcodes defining MPX instructions operated as NOP, and the CALL, RET, JMP, and Jcc instructions ignored any F2H prefix.**
2. Bit 8 and bits 13:11 correspond to supervisor state components. Since bits can be set in XCR0 only for user state components, those bits of XCR0 must be 0.

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

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

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
 - If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.
 - If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.
2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX state and software can execute MPX instructions. If XCR0[7:5] is 111b, the XSAVE feature set can be used to manage AVX-512 state and software can execute AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

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

- IA32_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32_XSS[8] = 1. The value of IA32_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32_XSS[8] = 0).
- IA32_XSS[12:11] are associated with CET state (see Section 13.5.8), IA32_XSS[11] with CET_U state and IA32_XSS[12] with CET_S state. Software can use the XSAVES and XRSTORS to manage CET_U state (respectively, CET_S state) only if IA32_XSS[11] = 1 (respectively, IA32_XSS[12] = 1). The value of IA32_XSS[12:11] does not determine whether software can use CET (the feature can be used even if either of IA32_XSS[12:11] is 0).
- IA32_XSS[13] is associated with HDC state (see Section 13.5.9). Software can use XSAVES and XRSTORS to manage HDC state only if IA32_XSS[13] = 1. The value of IA32_XSS[13] does not determine whether software can use hardware duty cycling (the feature can be used even if IA32_XSS[13] = 0).
- IA32_XSS[16] is associated with HWP state (see Section 13.5.10). Software can use XSAVES and XRSTORS to manage HWP state only if IA32_XSS[16] = 1. The value of IA32_XSS[16] does not determine whether software can use hardware P-states (the feature can be used even if IA32_XSS[16] = 0).
- IA32_XSS[63:17], IA32_XSS[15:14], IA32_XSS[10:9] and IA32_XSS[7:0] are reserved.¹ Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32_XSS MSR is 0 coming out of RESET.

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

13.4 XSAVE AREA

The XSAVE feature set includes instructions that save and restore the XSAVE-managed state components to and from memory: XSAVE, XSAVEOPT, XSAVEC, and XSAVES (for saving); and XRSTOR and XRSTORS (for restoring). The processor organizes the state components in a region of memory called an **XSAVE area**. Each of the save and

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

restore instructions takes a memory operand that specifies the 64-byte aligned base address of the XSAVE area on which it operates.

Every XSAVE area has the following format:

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

13.4.1 Legacy Region of an XSAVE Area

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

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

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

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
XMM11															336	
XMM12															352	
XMM13															368	
XMM14															384	
XMM15															400	

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. The XSAVE feature set does not use bytes 511:416; bytes 463:416 are reserved. Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the legacy region of an XSAVE area.

13.4.2 XSAVE Header

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

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called **XSTATE_BV**. It identifies the state components in the XSAVE area.
- Bytes 15:8 of the XSAVE header is a state-component bitmap called **XCOMP_BV**. It is used as follows:
 - XCOMP_BV[63] indicates the format of the extended region of the XSAVE area (see Section 13.4.3). If it is clear, the standard format is used. If it is set, the compacted format is used; XCOMP_BV[62:0] provide format specifics as specified in Section 13.4.3.
 - XCOMP_BV[63] determines which form of the XRSTOR instruction is used. If the bit is set, the compacted form is used; otherwise, the standard form is used. See Section 13.8.
 - All bits in XCOMP_BV should be 0 if the processor does not support the compaction extensions to the XSAVE feature set.
- Bytes 63:16 of the XSAVE header are reserved.

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

13.4.3 Extended Region of an XSAVE Area

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

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

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

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

- **Standard format.** Each state component *i* ($i \geq 2$) is located at the byte offset from the base address of the XSAVE area enumerated in CPUID.(EAX=0DH,ECX=*i*):EBX. (CPUID.(EAX=0DH,ECX=*i*):EAX enumerates the number of bytes required for state component *i*).
- **Compacted format.** Each state component *i* ($i \geq 2$) is located at a byte offset from the base address of the XSAVE area based on the XCOMP_BV field in the XSAVE header:

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

13.5 XSAVE-MANAGED STATE

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

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

13.5.1 x87 State

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

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

- If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
- Bytes 23:22 are not used.
 - If the instruction has a REX prefix with REX.W = 1, bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers ST0–ST7 (MM0–MM7). Each of the 8 register is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

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

13.5.2 SSE State

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

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.²
- Bytes 31:28 are used for the MXCSR_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

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

13.5.3 AVX State

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

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

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

The XSAVE feature set partitions YMM0_H–YMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are used for YMM0_H–YMM7_H. Bytes 255:128 are used for

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

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

YMM8_H–YMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8_H–YMM15_H. See Section 13.13. In general, bytes $16i+15:16i$ are used for YMM i _H (for $0 \leq i \leq 15$).

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

13.5.4 MPX State

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

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

- **BNDREGS state.**
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREGS state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=3):EAX enumerates the size (in bytes) required for BNDREGS state. The BNDREGS section is used for the 4 128-bit bound registers BND0–BND3, with bytes $16i+15:16i$ being used for BND i .
- **BNDCSR state.**
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=4):EBX enumerates the offset of the section of the extended region of the XSAVE area used for BNDCSR state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=4):EAX enumerates the size (in bytes) required for BNDCSR state. In the BNDCSR section, bytes 7:0 are used for BNDCFGU and bytes 15:8 are used for BNDSTATUS.

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

13.5.5 AVX-512 State

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

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

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

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

- **Opmask state.**
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes)

required for opmask state. The opmask section is used for the 8 64-bit opmask registers k0–k7, with bytes $8i+7:8i$ being used for ki .

- **ZMM_Hi256 state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM_Hi256 state.

The XSAVE feature set partitions ZMM0_H–ZMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM_Hi256-state section are used for ZMM0_H–ZMM7_H.

Bytes 511:256 are used for ZMM8_H–ZMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8_H–ZMM15_H. See Section 13.13. In general, bytes $32i+31:32i$ are used for ZMM i _H (for $0 \leq i \leq 15$).

- **Hi16_ZMM state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16_ZMM state.

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

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

13.5.6 PT State

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

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

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

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

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

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

13.5.7 PKRU State

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

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

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

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

13.5.8 CET State

The register state used by Control-Flow Enforcement Technology (CET) comprises the two 64-bit MSRs (IA32_U_CET and IA32_PL3_SSP) that manage CET when CPL = 3 (**CET_U state**); and the three 64-bit MSRs (IA32_PL0_SSP–IA32_PL2_SSP) that manage CET when CPL < 3 (**CET_S state**). Together, these two user state components compose **CET state**.¹

As noted in Section 13.1, the XSAVE feature set manages CET state as supervisor state components 11–23. Thus, CET state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **CET_U state.**
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=11):EAX enumerates the size (in bytes) required for CET_U state. The CET_U section is used for the 64-bit MSRs IA32_U_CET and IA32_PL3_SSP, with bytes 7:0 being used for IA32_U_CET and bytes 15:8 being used for IA32_PL3_SSP.
- **CET_S state.**
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=12):EAX enumerates the size (in bytes) required for CET_S state. The CET_S section is used for the three 64-bit MSRs IA32_PL0_SSP–IA32_PL2_SSP, with bytes $8i+7:8i$ being used for IA32_PL i _SSP.

The two components of CET state are XSAVE-managed and CET is not XSAVE-enabled. The XSAVE feature set can operate on CET_U state (respectively, CET_S state) only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage CET_U state (respectively, CET_S state) by setting IA32_XSS[11] (respectively, IA32_XSS[12]). Software can otherwise use CET and access the CET MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage CET state.

13.5.9 HDC State

The register state used by hardware duty cycling (**HDC state**) comprises the IA32_PM_CTL1 MSR.

As noted in Section 13.1, the XSAVE feature set manages HDC state as supervisor state component 13. Thus, HDC state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=13):EAX enumerates the size (in bytes) required for HDC state. The IA32_PM_CTL1 MSR is allocated 8 bytes at byte offset 0 in the state component.

HDC state is XSAVE-managed but hardware duty cycling is not XSAVE-enabled. The XSAVE feature set can operate on HDC state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HDC state

1. The IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs also control CET when CPL < 3. However, they are not managed by the XSAVE feature set and are thus not considered in this chapter.

(IA32_XSS[13] = 1). Software can otherwise use hardware duty cycling and access the IA32_PM_CTL1 MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HDC state.

13.5.10 HWP State

The register state used by hardware P-states (**HWP state**) comprises the IA32_HWP_REQUEST MSR.

As noted in Section 13.1, the XSAVE feature set manages HWP state as supervisor state component 16. Thus, HWP state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=16):EAX enumerates the size (in bytes) required for HWP state. The IA32_HWP_REQUEST MSR is allocated 8 bytes at byte offset 0 in the state component.

HWP state is XSAVE-managed but the hardware P-states feature is not XSAVE-enabled. The XSAVE feature set can operate on HWP state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HWP state (IA32_XSS[16] = 1). Software can otherwise use hardware P-states and access the IA32_HWP_REQUEST MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HWP state.

13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

The XSAVEOPT, XSAVEC, and XSAVES instructions use two optimizations to reduce the amount of data that they write to memory. They avoid writing data for any state component known to be in its initial configuration (the **init optimization**). In addition, if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the **modified optimization**). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.) The operation of XSAVEOPT, XSAVEC, and XSAVES are described in more detail in Section 13.9 through Section 13.11.

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

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

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

Executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. Such an execution of XGETBV always sets EAX[1] to 1 if XCR0[1] = 1 and MXCSR does not have its RESET value of 1F80H. Section 13.2 explains how software can determine whether a processor supports this use of XGETBV.

- XMODIFIED denotes the state-component bitmap corresponding to the modified optimization. If $XMODIFIED[i] = 0$, state component i is known not to have been modified since the most recent execution of XRSTOR or XRSTORS; otherwise $XMODIFIED[i] = 1$. It is possible for $XMODIFIED[i]$ to be 1 even when state component i has not been modified since the most recent execution of XRSTOR or XRSTORS. On a processor that does not support the modified optimization, $XMODIFIED[i]$ is always 1 for every value of i .

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

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

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

- **x87 state.** x87 state is in its initial configuration if the following all hold: FCW is 037FH; FSW is 0000H; FTW is FFFFH; FCS and FDS are each 0000H; FIP and FDP are each 00000000_00000000H; each of ST0–ST7 is 0000_00000000_00000000H.
- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM15 is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM7 is 0. XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update XMM8–XMM15. (See Section 13.13.)
- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM15_H is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM7_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update YMM8_H–YMM15_H. (See Section 13.13.)
- **BNDREGS state.** BNDREGS state is in its initial configuration if the value of each of BND0–BND3 is 0.
- **BNDCSR state.** BNDCSR state is in its initial configuration if BNDCFGU and BNDCSR each has value 0.
- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers k0–k7 is 0.
- **ZMM_Hi256 state.** In 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM15_H is 0. Outside 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM7_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM8_H–ZMM15_H. (See Section 13.13.)
- **Hi16_ZMM state.** In 64-bit mode, Hi16_ZMM state is in its initial configuration if each of ZMM16–ZMM31 is 0. Outside 64-bit mode, Hi16_ZMM state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM31–ZMM31. (See Section 13.13.)
- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.
- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.
- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.
- **CET_U state.** CET_U state is in its initial configuration if both of the MSRs are 0.
- **CET_S state.** CET_S state is in its initial configuration if each of the three MSRs is 0.
- **HDC state.** HDC state is in its initial configuration if the value of the IA32_PM_CTL1 MSR is 1.
- **HWP state.** HWP state is in its initial configuration if the value of the IA32_HWP_REQUEST MSR is 8000FF01H.

13.7 OPERATION OF XSAVE

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

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

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

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

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

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

- If state component i is in its initial configuration, $XINUSE[i]$ may be either 0 or 1, and $XSTATE_BV[i]$ may be written with either 0 or 1.
 $XINUSE[1]$ pertains only to the state of the XMM registers and not to MXCSR. Thus, $XSTATE_BV[1]$ may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
- If state component i is not in its initial configuration, $XINUSE[i] = 1$ and $XSTATE_BV[i]$ is written with 1.
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

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

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

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

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

13.8 OPERATION OF XRSTOR

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

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

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

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

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

13.8.1 Standard Form of XRSTOR

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

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

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

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

- If $XSTATE_BV[i] = 0$, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.
The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR.
- If $XSTATE_BV[i] = 1$, the state component is loaded with data from the XSAVE area. See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.
State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i , $2 \leq i \leq 62$, is located in the extended region; the standard form of XRSTOR uses the standard format for the extended region (see Section 13.4.3).

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

13.8.2 Compacted Form of XRSTOR

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

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

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

- If $XSTATE_BV[i] = 0$, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.
If $XSTATE_BV[1] = 0$, the compacted form XRSTOR initializes MXCSR to 1F80H. (This differs from the standard form of XRSTOR, which loads MXCSR from the XSAVE area whenever either $RFBM[1]$ or $RFBM[2]$ is set.)
State component i is set to its initial configuration as indicated above if $RFBM[i] = 1$ and $XSTATE_BV[i] = 0$ — **even if XCOMP_BV[i] = 0**. This is true for all values of i , including 0 (x87 state) and 1 (SSE state).
- If $XSTATE_BV[i] = 1$, the state component is loaded with data from the XSAVE area.¹ See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.
State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i , $2 \leq i \leq 62$, is located in the extended region; the compacted form of the XRSTOR instruction uses the compacted format for the extended region (see Section 13.4.3).

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

13.8.3 XRSTOR and the Init and Modified Optimizations

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

- The processor updates its tracking for the init optimization as follows:

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

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

13.9 OPERATION OF XSAVEOPT

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

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

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

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

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

- If $RFBM[i] = 0$, $XSTATE_BV[i]$ is not changed.
- If $RFBM[i] = 1$, $XSTATE_BV[i]$ is set to the value of $XINUSE[i]$. Section 13.6 defines $XINUSE$ to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
 - If the state component is in its initial configuration, $XINUSE[i]$ may be either 0 or 1, and $XSTATE_BV[i]$ may be written with either 0 or 1.
 $XINUSE[1]$ pertains only to the state of the XMM registers and not to MXCSR. Thus, $XSTATE_BV[1]$ may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
 - If the state component is not in its initial configuration, $XSTATE_BV[i]$ is written with 1.
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

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

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

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

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

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

- **Init optimization.**

If $XINUSE[i] = 0$, state component i is not saved to the XSAVE area (even if $RFBM[i] = 1$). (See below for exceptions made for MXCSR.)

- **Modified optimization.**

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

- $w = CPL$;
- $x = 1$ if and only if the logical processor is in VMX non-root operation;
- y is the linear address of the XSAVE area being used by XSAVEOPT; and
- z is 00000000_00000000H. (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of XRSTOR used the compacted form, or if an execution of XRSTORS followed the last execution of XRSTOR.)

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

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

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

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

13.10 OPERATION OF XSAVEC

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

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

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

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

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

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE_BV[i] ($0 \leq i \leq 63$) as follows:¹

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

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

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

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

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

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

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If XINUSE[i] = 0, state component i is not saved to the XSAVE area (even if RFBM[i] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC saves all of state component 1 (SSE — including the XMM registers) even if XINUSE[1] = 0. Unlike the XSAVE instruction, RFBM[2] does not determine whether XSAVEC saves MXCSR and MXCSR_MASK.

13.11 OPERATION OF XSAVES

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

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

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

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

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

- If RFBM[i] = 0, XSTATE_BV[i] is written as 0.

1. Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does **not** read the XSTATE_BV field of the XSAVE header.

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

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

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

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

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

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

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

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

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

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

13.12 OPERATION OF XRSTORS

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

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

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

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

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

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

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

- If XSTATE_BV[i] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If XSTATE_BV[1] = 0, XRSTORS initializes MXCSR to 1F80H.
State component i is set to its initial configuration as indicated above if RFBM[i] = 1 and XSTATE_BV[i] = 0 — **even if XCOMP_BV[i] = 0**. This is true for all values of i , including 0 (x87 state) and 1 (SSE state).
- If XSTATE_BV[i] = 1, the state component is loaded with data from the XSAVE area.¹ See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XRSTORS. See Section 13.13 for details regarding faults caused by memory accesses.

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

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

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

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

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

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

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

13.13 MEMORY ACCESSES BY THE XSAVE FEATURE SET

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

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

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

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

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

5. Updates to Chapter 18, Volume 1

Updates to Chapter 18 added to the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

Changes to this chapter: Typo correction and additional information added to section 18.2 "Shadow Stacks".

18.1 INTRODUCTION

Return-oriented programming (ROP), and similarly CALL/JMP-oriented programming (COP/JOP), have been the prevalent attack methodologies for stealth exploit writers targeting vulnerabilities in programs. These attack methodologies have the common elements:

- A code module with execution privilege and contain small snippets of code sequence with the characteristic: at least one instruction in the sequence being a control transfer instruction that depends on data either in the return stack or in a register for the target address.
- Diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register).

Control-Flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/COP/JOP style control-flow subversion attacks:

- Shadow stack: Return address protection to defend against ROP.
- Indirect branch tracking: Free branch protection to defend against COP/JOP.

Both capabilities introduce new instruction set extensions, and are described in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

Control-Flow Enforcement Technology introduces a new exception (#CP) with interrupt vector 21.

18.1.1 Shadow Stack

A shadow stack is a second stack for the program that is used exclusively for control transfer operations. This stack is separate from the data stack and can be enabled for operation individually in user mode or supervisor mode. When shadow stacks are enabled, the CALL instruction pushes the return address on both the data and shadow stack. The RET instruction pops the return address from both stacks and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP). Note that the shadow stack only holds the return addresses and not parameters passed to the call instruction.

The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table protections are extended to support an additional attribute for pages to mark them as "Shadow Stack" pages. When shadow stacks are enabled, control transfer instructions/flows like near call, far call, call to interrupt/exception handlers, etc. store return addresses to the shadow stack and the access will fault if the underlying page is not marked as a "Shadow Stack" page. However stores from instructions like MOV, XSAVE, etc. will not be allowed. Likewise control transfer instructions like near RET, far RET, IRET, etc. when they attempt to read from the shadow stack the access will fault if the underlying page is not marked as a "Shadow Stack" page. This paging protection detects and prevents conditions that cause an overflow or underflow of the shadow stack when the shadow stack is delimited by non-shadow stack guard pages, or any malicious attempts to redirect the processor to consume data from addresses that are not shadow stack addresses.

18.1.2 Indirect Branch Tracking

The ENDBRANCH instruction is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect JMP and CALL instructions. When one of these instructions is executed, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program

stream must be an ENDBRANCH. If the next instruction is not an ENDBRANCH, the processor causes a control protection exception (#CP); otherwise, the state machine moves back to IDLE state.

18.1.3 Speculative Behavior when CET is Enabled

Speculative execution of near indirect JMP/CALL/RET indirect branches may be able to create an active side channel vulnerability that reveals the contents of data.

There are two basic methods that an attacker may be able to use to control indirect branch speculation in order to speculatively execute code that causes a side channel:

1. Attacker controlled prediction.
2. Attacker controlled jump redirection.

With attacker controlled prediction, the attacker trains indirect branch predictors such that the desired victim indirect branch goes to the attacker desired location. Examples include Branch Target Injection (also called "Variant 2" and "Spectre") and RSB wrap on underflow (also called "ret2spec").

With attacker controlled jump redirection, the attacker controls a speculative-only value used as input to the indirect branch so that the branch mispredicts to the attacker desired location. Examples of this include Bound Check Bypass Store (where a speculative store containing an attacker controlled value may overwrite the indirect branch target before the load of the target) and Speculative Store Bypass (where a load of the indirect branch target may bypass the most recent store of the target value and thus speculatively read an older attacker controlled value at the same memory location).

In addition to the existing mitigation features like IBRS, STIBP and IBPB, processors supporting CET will have a variety of additional features to constrain control flow speculation in order to mitigate such attacks. For details on these features, see Section 18.2.6, "Constraining Execution at Targets of RET" and Section 18.3.8, "Constraining Speculation after Missing ENDBRANCH".

18.2 SHADOW STACKS

A shadow stack is a second expand down stack used exclusively for control transfer operations. This stack is separate from the data stack. The shadow stack is not used to store data and hence is not explicitly writeable by software. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions. The shadow stack feature can be enabled separately in user mode (CPL == 3) or supervisor mode (CPL < 3).

Shadow stacks operate only in protected mode. Shadow stacks cannot be enabled in virtual 8086 mode.

It is recommended to not configure the shadow stack in the linear address range 0 to 64 KB or adjacent to the canonical address boundary.

18.2.1 Shadow Stack Pointer and its Operand and Address Size Attributes

When CET is enabled the processor supports a new architectural register, shadow stack pointer (SSP), when the processor supports the shadow stack feature. The SSP cannot be directly encoded as a source, destination or memory operand in instructions. The SSP points to the current top of the shadow stack.

The width of the shadow stack is 32-bit in 32-bit/compatibility mode and is 64-bit in 64-bit mode. The address-size attribute of the shadow stack is likewise 32-bit in 32-bit/compatibility mode and 64-bit in 64-bit mode.

18.2.2 Terminology

When shadow stacks are enabled, certain control transfer instructions/flows and shadow stack management instructions do loads/stores to the shadow stack. Such load/stores from control transfer instructions and shadow stack management instructions are termed as shadow_stack_load and shadow_stack_store to distinguish them from a load/store performed by other instructions like MOV, XSAVES, etc.

The pseudocode for the instruction operations use the notation `ShadowStackEnabled(CPL)` as a test of whether shadow stacks are enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

```
ShadowStackEnabled(CPL):
  IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
    IF CPL = 3
      THEN
        (* Obtain the shadow stack enable from IA32_U_CET MSR (MSR address 6A0H) used to enable
           feature for CPL = 3 *)
        SHADOW_STACK_ENABLED = IA32_U_CET.SH_STK_EN;
      ELSE
        (* Obtain the shadow stack enable from IA32_S_CET MSR (MSR address 6A2H) used to enable
           feature for CPL < 3 *)
        SHADOW_STACK_ENABLED = IA32_S_CET.SH_STK_EN;
    FI;
    IF SHADOW_STACK_ENABLED = 1
      THEN
        return TRUE;
      ELSE
        return FALSE;
    FI;
  ELSE
    (* Shadow stacks not enabled in real mode and virtual-8086 mode or if the master CET feature
       enable in CR4 is disabled *)
    return FALSE;
  ENDIF
```

Additionally, the following terms are used.

- `ShadowStackPush4B`: Decrements the shadow stack pointer (SSP) by 4 bytes and copies the 4 byte source operand to the top of the shadow stack.
- `ShadowStackPush8B`: Decrements the shadow stack pointer (SSP) by 8 bytes and copies the 8 byte source operand to the top of the shadow stack.
- `ShadowStackPop4B`: Copies 4 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 4 bytes to point to the new top of stack.
- `ShadowStackPop8B`: Copies 8 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 8 bytes to point to the new top of stack.
- `shadow_stack_lock_cmpxchg8B(address, new_value, expected_value)`: this function executes atomically and compares the `expected_value` to the 8 byte read from memory specified by the `address` operand using a locked `shadow_stack_load`. If the two values are equal, the `new_value` is written to memory specified by the `address` operand using a locked `shadow_stack_store`. If the two values are not equal, then the value read by the locked `shadow_stack_load` is written back. The memory specified by the `address` operand receives a write cycle without regard to the result of the comparison. The function returns the value read from the memory specified by the `address` operand.

18.2.3 Supervisor Shadow Stack Token

On an inter-privilege far CALL or when calling an interrupt/exception handler at a higher privilege level, a stack switch occurs; if shadow stacks are enabled at the new privilege level, then a shadow stack switch occurs. Shadow stacks that can be switched to by hardware as part of a privilege change are required to have a supervisor shadow stack token set up by the supervisor to provide the address of the new SSP register. The supervisor shadow stack tokens also serve the purpose of enforcing that a shadow stack can be made active on only one logical processor

when switched to by the processor. The supervisor shadow stack token must be set up only on shadow stacks intended to be used on these transfers.

The supervisor shadow stack token is a 64-bit value formulated as follows.

- Bit 63:3: Bits 63:3 of the linear address of the supervisor shadow stack token.
- Bit 2: Reserved. Must be zero.
- Bit 1: Reserved. Must be zero.
- Bit 0: Busy bit. If 0, indicates this shadow stack is not active on any logical processor. If 1, indicates this shadow stack is currently active on one of the logical processors.

The following figure illustrates a supervisor shadow stack with a supervisor shadow stack token located at its base.

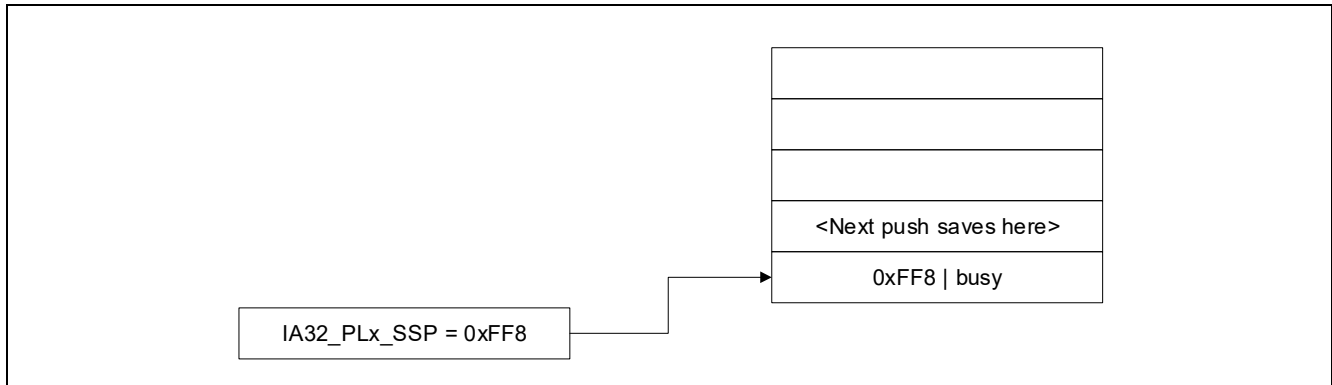


Figure 18-1. Supervisor Shadow Stack with a Supervisor Shadow Stack Token

The WRMSR instruction ensures that the address specified in the IA32_PLx_SSP MSR (where $0 \leq x \leq 3$) is required to be 4 byte aligned; otherwise, the instruction causes a general protection exception (#GP(0)). The processor does the following checks prior to switching to a supervisor shadow stack programmed into the IA32_PLx_SSP MSR. These steps are performed atomically.

1. Load the supervisor shadow stack token from the address specified in the IA32_PLx_SSP MSR using a `shadow_stack_load`.
2. Check if the busy bit in the token is 0; reserved bits must be 0.
3. Check if the address programmed in the MSR matches the address in the supervisor shadow stack token; reserved bits must be 0.
4. If checks 2 and 3 are successful, then set the busy bit in the token using a `shadow_stack_store` and switch the SSP to the value specified in the IA32_PLx_SSP MSR.
5. If checks 2 or 3 fail, then the busy bit is not set and a #GP(0) exception is raised.

On a far RET, the instruction clears the busy bit in the shadow stack token as follows. These steps are also performed atomically.

1. Load the supervisor shadow stack token from the SSP using a `shadow_stack_load`.
2. Check if the busy bit in the token is 1; reserved bits must be 0 and reserved bits must be 0.
3. Check if the address programmed in supervisor shadow stack token matches SSP; reserved bits must be 0.
4. If checks 2 and 3 are successful, then clear the busy bit in the token using a `shadow_stack_store`; else continue without modifying the contents of the shadow stack pointed to by SSP.

The operations described here are also applicable to a far transfer performed when calling an interrupt or exception handler through an interrupt/trap gate in the IDT. Likewise, the IRET instruction behaves similar to the Far RET instruction.

18.2.4 Shadow Stack Usage on Task Switch

A task switch (see Chapter 7, “Task Management” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*) may be invoked by:

- JMP or CALL instruction to a TSS descriptor in the GDT.
- JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.

With shadow stack enabled, the new task must be associated with a 32-bit TSS and must not be in virtual-8086 mode. The 32-bit SSP for the new task is located at offset 104 in the 32-bit TSS. Thus the TSS of the new task must be at least 108 bytes. This SSP is required to be 8 byte aligned, and required to point to a “supervisor shadow stack” token (though the task may be at CPL3).

On a task switch initiated by a CALL instruction, an interrupt, or exception, the SSP of the old task is pushed onto the shadow stack of the new task along with the CS and LIP of the old task. This is true even for a nested task switch initiated by a CALL instruction. Likewise, on a task switch initiated by IRET, the SSP of the new task is restored from the shadow stack of old task. The CS and LIP on the shadow stack of the old task are matched against the return address determined by the CS and EIP of the new task. If the match fails, a #CP(FAR-RET/IRET) exception is reported.

18.2.5 Switching Shadow Stacks

The architecture provides a mechanism to switch shadow stacks using a pair of instructions; RSTORSSP and SAVEPREVSSP. The RSTORSSP instruction verifies a shadow-stack-restore token located at the top of the new shadow stack and referenced by the memory operand of this instruction. After RSTORSSP determines the validity of the restore point on the new shadow stack, it switches the SSP to point to the token. The shadow-stack-restore token is a 64-bit value formatted as follows.

- Bit 63:2: Value of shadow stack pointer when this restore point was created.
- Bit 1: Reserved. Must be zero.
- Bit 0: Mode bit. If 0, the token is a compatibility/legacy mode shadow-stack-restore token. If 1, then this shadow stack restore token can be used with a RSTORSSP instruction in 64-bit mode.

The shadow-stack-restore token is created by the SAVEPREVSSP instruction. The operating system may also create a restore point on a shadow stack by creating a shadow-stack-restore token.

Once the shadow stack has been switched to a new shadow stack by the RSTORSSP instruction, software can create a restore point on the old shadow stack by executing the SAVEPREVSSP instruction. In order to allow the SAVEPREVSSP instruction to determine the address where to save the shadow-stack-restore token, the RSTORSSP instruction replaces the shadow-stack-restore token with a previous-ssp token that holds the value of the SSP at the time the RSTORSSP instruction was invoked. The previous-ssp token is formatted as follows.

- Bit 63:2: Shadow stack pointer when the RSTORSSP instruction was invoked, i.e., the SSP of the old shadow stack.
- Bit 1: Set to 1.
- Bit 0: Mode bit. If 0, then this previous-ssp token can be used with a SAVEPREVSSP instruction in compatibility/legacy mode. If 1, then this previous-ssp token can be used with a SAVEPREVSSP instruction in 64-bit mode.

The following figure illustrates the RSTORSSP instruction operation during a shadow stack switching sequence.

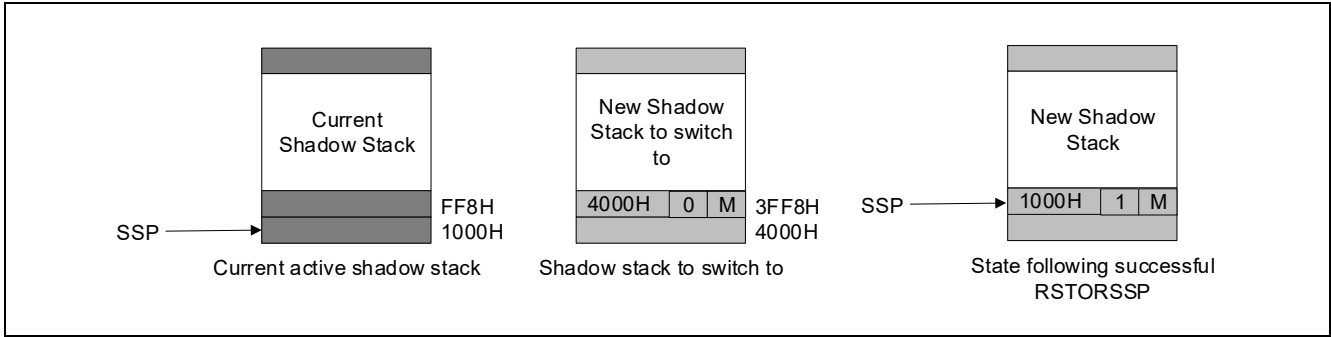


Figure 18-2. RSTORSSP to Switch to New Shadow Stack

In this example, the initial SSP is 1000H and the shadow-stack-restore token is on a new shadow stack at address 3FF8H. The token at address 3FF8H holds the SSP when this restore point was created; in this example it is 4000H.

In order to switch to the new shadow stack, the RSTORSSP instruction is invoked with the memory operand pointing set to 3FF8H. When the RSTORSSP instruction completes, the SSP is set to 3FF8H and the shadow-stack-restore token at 3FF8H is replaced by a previous-ssp token that holds the address 1000H, i.e., the old SSP.

The following figure illustrates the SAVEPREVSSP instruction operation during a shadow stack switching sequence.

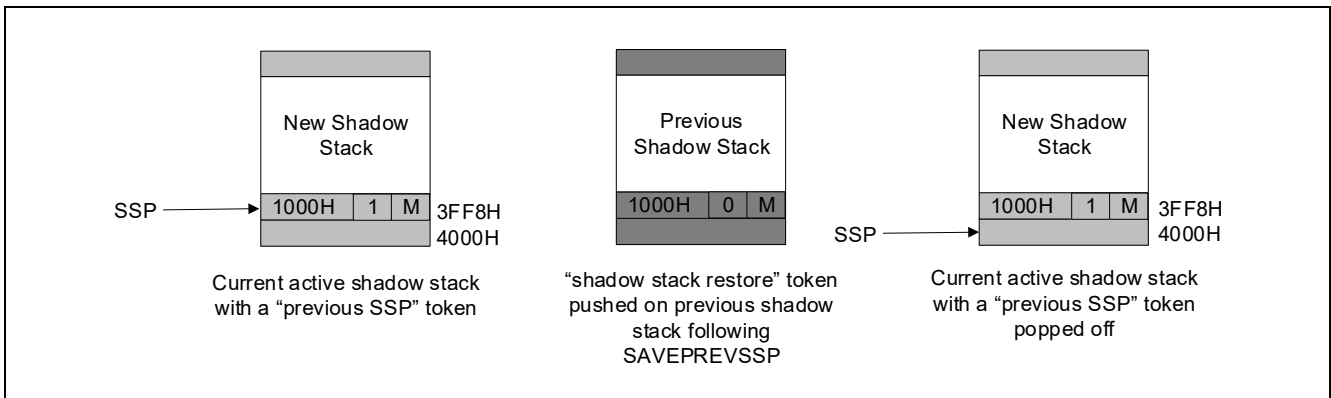


Figure 18-3. SAVEPREVSSP to Save a Restore Point

To allow switching back to this old shadow stack, a SAVEPREVSSP instruction is now invoked. The SAVEPREVSSP instruction does not take any memory operand and expects to find a previous-ssp token at the top of the shadow stack, i.e., at address 3FF8H. The SAVEPREVSSP instruction then saves a shadow-stack-restore token on the old shadow stack at address FF8H, and the token itself holds the address 1000H which is the address recorded in the previous-ssp token. The SAVEPREVSSP instruction also pops the previous-ssp token off the current shadow stack and thus the SSP following SAVEPREVSSP is 4000H.

Subsequently to switch back to the old shadow stack, a RSTORSSP instruction may be invoked with memory operand set to FF8H.

If, following a switch to a new shadow stack, it is not required to create a restore point on the old shadow stack, then the previous-ssp token created by the RSTORSSP instruction can be popped off the shadow stack by using the INCSSP instruction.

See the SAVEPREVSSP and RSTORSSP instruction operations for the detailed algorithm.

18.2.6 Constraining Execution at Targets of RET

Instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (either from normal stack or shadow stack) are speculative-only or do not match, unless the target of the RET is also predicted (e.g., by a Return Stack Buffer prediction), when CET shadow stack is enabled. A RET address would be speculative-only if it was modified by an older speculative-only store, or was an older value than the most recent value stored to that address on the logical processor.

18.3 INDIRECT BRANCH TRACKING

When the indirect branch tracking feature is active, the indirect JMP/CALL instruction behavior changes as follows.

- **JMP:** If the next instruction retired after an indirect JMP is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, then a #CP fault is generated. Below JMP instructions are tracked to enforce an ENDBRANCH. Note that Jcc, RIP relative, and far direct JMP are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
 - JMP r/m16, JMP r/m32, JMP r/m64
 - JMP m16:16, JMP m16:32, JMP m16:64
- **CALL:** If the next instruction retired after an indirect CALL is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 in 64-bit mode, then a #CP fault is generated. Below CALL instructions are tracked to enforce an ENDBRANCH. Note that relative and zero displacement forms of CALL instructions are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
 - CALL r/m16, CALL r/m32, CALL r/m64
 - CALL m16:16, CALL m16:32, CALL m16:64

The ENDBR32 and ENDBR64 instructions will have the same effect as the NOP instruction on Intel 64 processors that do not support CET. On processors supporting CET, these instructions do not change register or flag state. This allows CET instrumented programs to execute on processors that do not support CET. Even when CET is supported and enabled, these NOP-like instructions do not affect the execution state of the program, do not cause any additional register pressure, and are minimally intrusive from power and performance perspectives.

The processor implements two dual-state machines to track indirect CALL/JMP for terminations. One state machine is maintained for user mode and one for supervisor mode. At reset the user and supervisor mode state machines are in IDLE state.

On instructions other than indirect CALL/JMP, the state machine stays in the IDLE state.

On an indirect CALL or JMP instruction, the state machine transitions to the WAIT_FOR_ENDBRANCH state.

In the WAIT_FOR_ENDBRANCH state, the indirect branch tracking state machine verifies the next instruction is an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, and either:

- Causes a #CP fault, or
- Allows the next instruction if legacy compatibility configuration allows (see Section 18.3.6).

The priority of the #CP(ENDBRANCH) exception relative to other events is as follows.

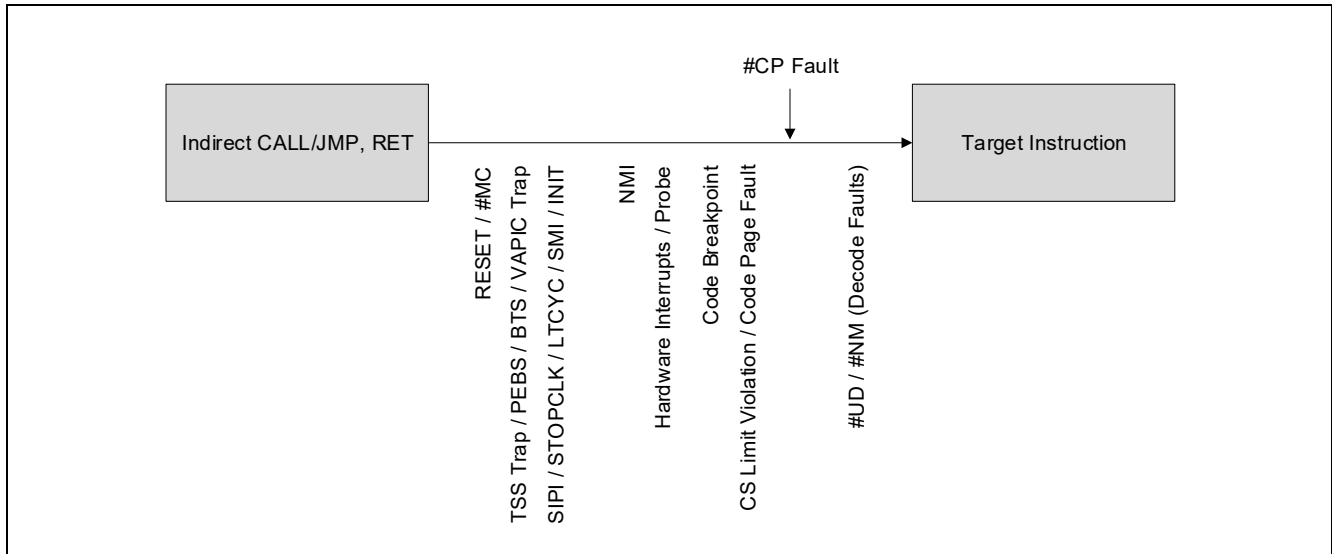


Figure 18-4. Priority of Control Protection Exception on Missing ENDBRANCH

Higher priority faults/traps/events that occur at the end of an indirect CALL/JMP are delivered ahead of any #CP(ENDBRANCH) fault. The CET state machine at the privilege level where the higher priority fault/trap/event occurred retains its state when the control transfers to the fault/trap/event handler. The instruction pointer pushed on the stack for a #CP(ENDBRANCH) fault is the address of the instruction at the target of the indirect CALL/JMP that caused the fault.

18.3.1 No-track Prefix for Near Indirect CALL/JMP

CET allows software to designate certain indirect CALL and JMP instructions as “non-tracked indirect control transfer instructions”. Software (e.g., compiler generated code for switch statements, jump tables, etc.) should use the no-track prefix only if they have generated code to validate the possible targets of this CALL/JMP to be legal targets. Software (e.g., compilers), when using the no-track prefix with CALL/JMP where an absolute offset is specified indirectly in a memory location, should ensure that such memory locations cannot be tampered. When enabled by setting the NO_TRACK_EN control in the IA32_U_CET/IA32_S_CET MSR, near indirect CALL and JMP instructions when prefixed with 3EH do not modify the CET indirect branch tracker. Far CALL and JMP instructions are always tracked and ignore the 3EH prefix. When this control is 0, near indirect CALL and JMP instructions are always tracked irrespective of the presence of the 3EH prefix.

In 64-bit mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix if there isn’t a 64H/65H prefix on the instruction.

In legacy/compatibility mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix when it is the last group 2 prefix on the instruction.

18.3.2 Terminology

The pseudocode for the instruction operations use a notation `EndbranchEnabled(CPL)` as a test of whether indirect branch tracking is enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

`EndbranchEnabled(CPL):`

```

IF CR4.CET = 1 AND CRO.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
    ELSE
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
  FI;
  IF ENDBR_ENABLED = 1
    THEN
      return TRUE;
    ELSE
      return FALSE;
  FI;
ELSE
  (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
  enable in CR4 is disabled *)
  return FALSE;
ENDIF

```

Likewise the notation `EndbranchEnabledAndNotSuppressed` is defined as follows:

`EndbranchEnabledAndNotSuppressed(CPL):`

```

IF CR4.CET = 1 AND CRO.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
      SUPPRESSED = IA32_U_CET.SUPPRESS;
    ELSE
      (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
      SUPPRESSED = IA32_S_CET.SUPPRESS;
  FI;
  IF ENDBR_ENABLED = 1 AND SUPPRESSED = 0
    THEN
      return TRUE;
    ELSE
      return FALSE;
  FI;
ELSE
  (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
  enable in CR4 is disabled *)
  return FALSE;
ENDIF

```

18.3.3 Indirect Branch Tracking

The hardware implements two CET indirect branch tracker state machines, one for user mode (CPL == 3) and one for supervisor mode (CPL < 3). At any time, which of the CET indirect branch trackers is in the active state depends on the CPL of the machine. When a user space program is executing, the CPL 3 CET indirect branch tracker is active. When supervisor mode software is executing, the CPL < 3 tracker is active. This section describes the various control transfer conditions and the tracker state on those transfers.

18.3.3.1 Control Transfers between CPL 3 and CPL < 3

Some events and instructions can cause control transfer to occur from CPL 3 to CPL < 3, and vice versa. As part of the CPL change the hardware also switches the active CET indirect branch tracker. For example, when an interrupt occurs during execution of a user mode (CPL == 3) program and it causes the CPL to switch to supervisor mode (CPL < 3) then, as part of the CPL change, the user mode CET indirect branch tracker becomes inactive and the supervisor mode CET indirect branch tracker becomes active. A subsequent IRET is used by the interrupt handler to return to the interrupted user mode program. This IRET causes the processor to switch the CPL to user mode (CPL == 3) and, as part of the CPL change, the supervisor mode CET indirect branch tracker becomes inactive and the user mode CET indirect branch tracker becomes active.

The CPL where the event or instruction that caused the control transfer occurs is termed the source CPL, and the CET indirect branch tracker state at that CPL is referred here as the source CET indirect branch tracker state. The CPL reached at the end of the control transfer is termed the destination CPL, and the CET indirect branch tracker state at that CPL is referred to as the destination CET indirect branch tracker state.

This section describes various cases of control transfers that occur between user mode (CPL 3) and supervisor mode (CPL < 3).

In all these cases the source CET indirect branch tracker state becomes not active and retains its state (IDLE, WAIT_FOR_ENDBRANCH), and the target CET indirect branch tracker state becomes active if there was no fault during the transfer.

- Case 1: Far CALL/JMPCALL/JMP, SYSCALL/SYSENTER
 - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH. This enforces that the subroutine invoked by a far CALL/JMPCALL/JMP must begin with an ENDBRANCH.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks
 - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
- Case 3: IRET
 - If indirect branch tracking enabled, the target indirect branch tracker becomes active and keeps its state. If the user mode was interrupted by a higher priority event, like an interrupt at the end of the indirect CALL/JMP, then when an IRET is used to return to the interrupted user mode program, the user mode indirect branch tracker retains its state and a #CP fault will occur if the next instruction decoded is not an endbr32/64 according to mode of machine.

18.3.3.2 Control Transfers within CPL 3 or CPL < 3

Some events and instructions can cause control transfer to occur within CPL 3 or CPL < 3. For such transfers since the CPL class does not change, the same indirect branch tracker is used at the beginning and end of the control transfer.

- Case 1: Far CALL/JMP, Near indirect CALL/JMPCALL/JMP
 - Far CALL/JMP: If indirect branch tracking is enabled, active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
 - Near indirect CALL/JMPCALL/JMP: If indirect branch tracking is enabled and not suppressed, active indirect branch tracker goes to WAIT_FOR_ENDBRANCH.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks

- If indirect branch tracking is enabled, the active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
- Case 3: IRET
 - If indirect branch tracking is enabled, the active indirect branch tracker keeps its state.

18.3.4 Indirect Branch Tracking State Machine

The state machine is described by Table 18-1.

Table 18-1. Indirect Branch Tracking State Machine

Current State	Trigger	Next State
TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1	Instructions other than indirect CALL/JMP or 3EH prefixed near indirect CALL/JMP and NO_TRACK_EN=1	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Indirect CALL/JMP without 3EH prefix Indirect CALL/JMP with 3EH prefix and NO_TRACK_EN=0 Far CALL/JMP	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1	INT3/INT1	TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	ENDBRANCH instruction	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Instructions other than ENDBRANCH, successful ENCLU[ERESUME] or INT3 or INT1	If legacy compatibility treatment is not enabled or if not allowed by legacy code page bitmap: <ul style="list-style-type: none"> ▪ No state change and deliver #CP (ENDBRANCH) If legacy compatibility treatment is enabled and transfer allowed by legacy code page bitmap: <ul style="list-style-type: none"> ▪ TRACKER=IDLE, SUPPRESS=ISUPPRESS_DIS, ENDBR_EN=1
TRACKER=x, SUPPRESS=x, ENDBR_EN=0	All instructions	TRACKER=x, SUPPRESS=x, ENDBR_EN=0
TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1	Far CALL/JMP, INTn/INT3/INTO	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	ENDBRANCH instruction Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	All other instructions including indirect CALL/JMP	TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1
TRACKER=1, SUPPRESS=1, ENDBR_EN=1 (This state cannot be reached by hardware and is disallowed as a valid state by WRMSR/XRSTORS/VM entry/VM exit)	NA	NA

18.3.5 INT3 Treatment

INT3 are treated special in the WAIT_FOR_ENDBRANCH state. Occurrence of INT3 do not move the tracker to IDLE but instead the #BP trap from the INT3 instructions respectively is delivered as a higher priority event than the #CP exception due to missing ENDBRANCH.

Inside an enclave, INT3 delivers a fault-class exception and thus does not require the CPL to be less than DPL in the IDT gate 3. Following opt-out entry, the instruction delivers #UD. Following opt-in entry, INT3 delivers #BP. The special treatment of INT3 in WAIT_FOR_ENDBRANCH state does not apply in enclave mode following opt-out entry.

18.3.6 Legacy Compatibility Treatment

ENDBRANCH legacy compatibility treatment allows a CET enabled program to be used with legacy software that was not compiled / instrumented with ENDBRANCH. A CET enabled program enters legacy compatibility treatment when all of the below conditions are met.

1. Legacy compatibility configuration is enabled in this CPL class by setting the LEG_IW_EN bit in IA32_U_CET/IA32_S_CET.
2. Control transfer is performed using an indirect CALL/JMP without no-track prefix to an instruction other than ENDBRANCH.
3. The legacy code page bitmap is setup to indicate that the target of the control transfer is a legacy code page.

The legacy code page bitmap is a data structure in program memory that is used by the hardware to determine if the code page to which a legacy transfer is being performed is allowed.

When a matching ENDBRANCH instruction is not decoded at the target of an indirect CALL/JMP when required, the processor performs the below actions.

CET indirect branch tracking state machine violation event handler:

If LEG_IW_EN == 1

LA = LIP;

IF ENCLAVE_MODE == 1

LA = LA - SECS.BASEADDR;

ENDIF

IF (EFER.LMA & CS.L) == 0

BITMAP_BYTE = load.Asiz_sylslnaddr. Osize8(BITMAP_BASE + LA[31:15])

ELSE

BITMAP_BYTE = load.Asiz_sylslnaddr. Osize8(BITMAP_BASE + LA[47:15])

FI;

IF BITMAP_BYTE & (1 << LA[14:12]) == 0 then Deliver #CP(ENDBRANCH) fault

IF CPL = 3

IA32_U_CET.TRACKER = IDLE

IA32_U_CET.SUPPRESS = IA32_U_CET.SUPPRESS_DIS == 0 ? 1 : 0

ELSE

IA32_S_CET.TRACKER = IDLE

IA32_S_CET.SUPPRESS = IA32_S_CET.SUPPRESS_DIS == 0 ? 1 : 0

ENDIF

Restart the instruction (handle all arch. consistency around MOV SS state machines, STI etc.) without opening up interrupt/trap window.

ELSE

Deliver #CP(ENDBRANCH) Fault

ENDIF

Faults/traps in pseudocode are delivered normally (e.g., #PF, EPT violation). On a fault, the active tracker holds the last value (WAIT_FOR_ENDBRANCH) and the address saved on the stack is the current IP (instruction that wasn't the ENDBRANCH).

The CET indirect branch tracking state machine is suppressed in legacy compatibility mode if the SUPPRESS_DIS control bit is 0.

Once the CET indirect branch tracking state machine has been suppressed, subsequent indirect CALL/JMP are not tracked for termination instruction.

Once CET indirect branch tracking has been suppressed, subsequent execution of ENDBRANCH instructions will do the following (see the ENDBR32 and ENDBR64 instructions in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for details).

```
IF EndbranchEnabled(CPL) == 0
    NOP
ELSE
    SUPPRESS = 0
    TRACKER = IDLE
ENDIF
```

18.3.6.1 Legacy Code Page Bitmap Format

The legacy code page bitmap is a flat bitmap whose linear address is pointed to by the EB_LEG_BITMAP_BASE. Each bit in the bitmap represents a 4K page in linear memory. If the bit is 1 it indicates that the corresponding code page is a legacy code page; else it is a CET-enabled code page.

The processor uses the linear address of the instruction to which legacy transfer was attempted to lookup the bitmap. Bits of the linear address used as index in the bitmap are as follows.

- In legacy and compatibility mode: Bits 31:12.
- In 64-bit mode (EFER.LMA=1 and CS.L=1): Bits 47:12.

18.3.7 Other Considerations

18.3.7.1 Intel® Transactional Synchronization Extensions (Intel® TSX) Interactions

The XBEGIN instruction encodes the relative offset to the abort handler and hence the fallback to the abort handler can be considered as a “direct” branch and the abort handler does not need to have an ENDBRANCH.

CET continues to enforce indirect CALL/JMP tracking within a transaction. Legacy compatibility treatment inside a transaction functions normally. If a transaction abort occurs then the processor sets the state of the indirect branch tracker to IDLE and not-suppressed.

18.3.7.2 #CP(ENDBRANCH) Priority w.r.t #NM and #UD

#NM, #UD and #CP(ENDBRANCH) are in the same priority class. Both #NM and #UD are opcode based faults. The #CP(ENDBRANCH) is prioritized higher than #NM and #UD as CET architecturally requires an ENDBRANCH at target of indirect CALL/JMP.

18.3.7.3 #CP(ENDBRANCH) Priority w.r.t #BP and #DB

Debug Exceptions priority is as follows.

- Traps delivered before any #CP(ENDBRANCH) fault: Data breakpoint trap, IO breakpoint trap single step trap, task switch trap.
- Code Breakpoint fault detected before instruction decode and delivered before #CP(ENDBRANCH).
- General-detect (GD) exception condition fault: Lower priority than #CP(ENDBRANCH).
- On IRET back from #DB/#BP, the source indirect branch tracker becomes active if enabled and not suppressed.

INT3 does not cause #CP(ENDBRANCH) to support debugger usage of replacing bytes of ENDBRANCH with INT3 to set breakpoints. INT3 at target of a CALL-JMP(indirect) cause #BP(INT3) instead of #CP(ENDBRANCH), #CP(ENDBRANCH) fault is delayed. #BP caused by INT3 treated like other events that are higher priority than CET fault. On IRET back from #BP the source indirect tracker becomes active if enabled and not suppressed.

18.3.8 Constraining Speculation after Missing ENDBRANCH

When the CET tracker is in the WAIT_FOR_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH.

This means that when indirect branch tracking is enabled and not suppressed, the instructions at the target of a near indirect JMP/CALL without the no-track prefix will only speculatively execute if there is an ENDBRANCH at the target. This can constrain both attacker controlled prediction as well as attacker controlled jump redirection attacks on near indirect JMPs/CALLs by reducing the gadgets available to an attacker using these techniques. Early implementations of CET may limit the speculative execution to a small number of instructions (less than 8, with no more than 5 loads) past a missing ENDBRANCH, while later implementations will completely block the speculative execution of instructions after a missing ENDBRANCH.

This mechanism also limits or blocks speculation of the next sequential instructions after an indirect JMP or CALL, presuming the JMP/CALL puts the CET tracker into the WAIT_FOR_ENDBRANCH state and the next sequential instruction is not an ENDBRANCH.

18.4 INTEL® TRUSTED EXECUTION TECHNOLOGY (INTEL® TXT) INTERACTIONS

GETSEC[ENTERACCS] and GETSEC[SENDER] clear CR4.CET, and it is not restored when these instructions complete.

■ GETSEC[EXITAC] will cause #GP(0) fault if CR4.CET is set.

6. Updates to Chapter 2, Volume 2A

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

Changes to this chapter: Typo corrections, update to Table 2-7 "RIP-Relative Addressing", and updates to Table 2-30 "EVEX Prefix Bit Field Functional Grouping" regarding inverted bit fields.

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

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

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

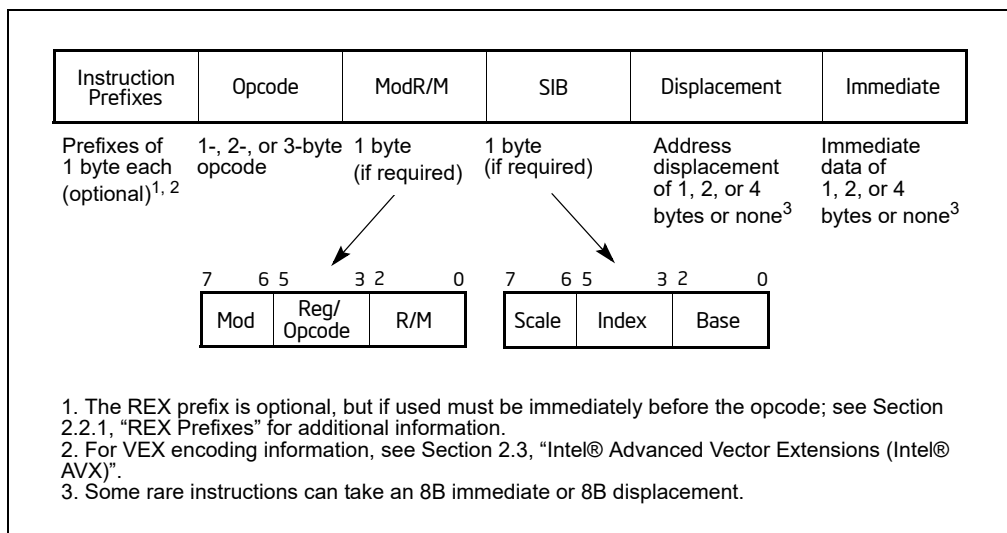


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

2.1.1 Instruction Prefixes

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

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

INSTRUCTION FORMAT

- BND prefix is encoded using F2H if the following conditions are true:
 - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
 - BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
 - When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Chapter 17, “Intel® MPX,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- Group 2
 - Segment override prefixes:
 - 2EH—CS segment override (use with any branch instruction is reserved).
 - 36H—SS segment override prefix (use with any branch instruction is reserved).
 - 3EH—DS segment override prefix (use with any branch instruction is reserved).
 - 26H—ES segment override prefix (use with any branch instruction is reserved).
 - 64H—FS segment override prefix (use with any branch instruction is reserved).
 - 65H—GS segment override prefix (use with any branch instruction is reserved).
 - Branch hints¹:
 - 2EH—Branch not taken (used only with Jcc instructions).
 - 3EH—Branch taken (used only with Jcc instructions).
- Group 3
 - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
 - 67H—Address-size override prefix.

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

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

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

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

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

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

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

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

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

2.1.2 Opcodes

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

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

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

For example, CVTQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

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

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

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

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

2.1.3 ModR/M and SIB Bytes

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

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

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

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

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

2.1.4 Displacement and Immediate Bytes

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

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

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

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

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

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

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute. If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

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

	Mod	11	
	RM		000
/digit (Opcode);	REG =	001	
	C8H	11	001000

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

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

			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP ¹ EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 ² [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8 ³ [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

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

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

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

NOTES:

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

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

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

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

NOTES:

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

MOD bits Effective Address

- 00 [scaled index] + disp32
01 [scaled index] + disp8 + [EBP]
10 [scaled index] + disp32 + [EBP]

2.2 IA-32E MODE

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

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

2.2.1 REX Prefixes

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

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

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

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

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

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

2.2.1.1 Encoding

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

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

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

2.2.1.2 More on REX Prefix Fields

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

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

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

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

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

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

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

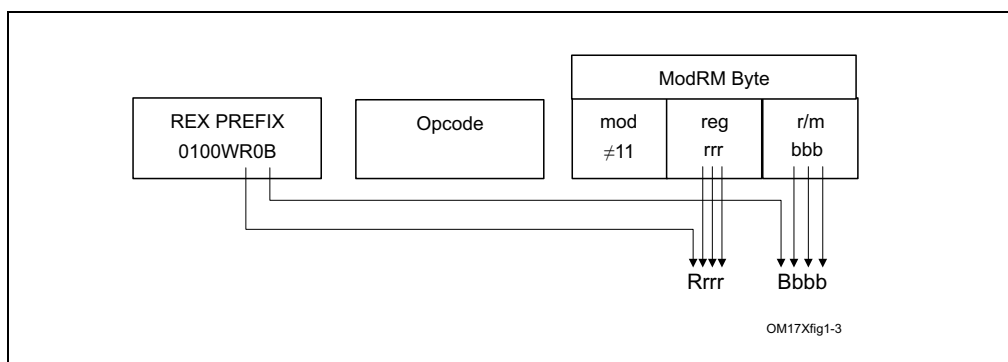


Figure 2-4. Memory Addressing Without a SIB Byte; REX.X Not Used

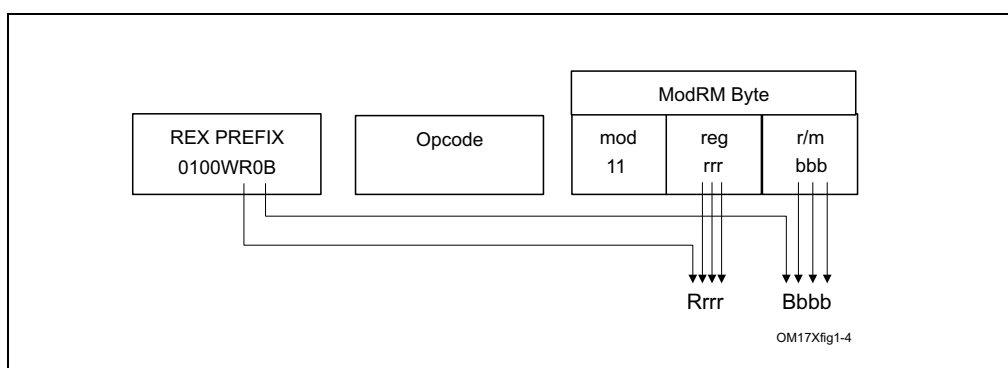


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

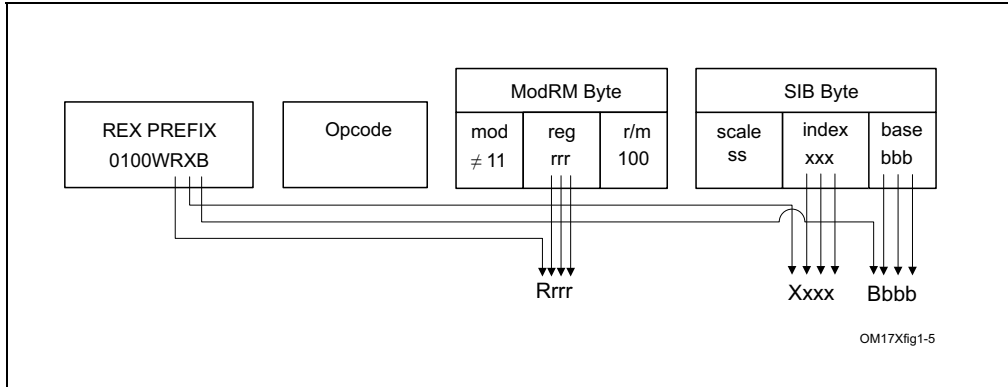


Figure 2-6. Memory Addressing With a SIB Byte

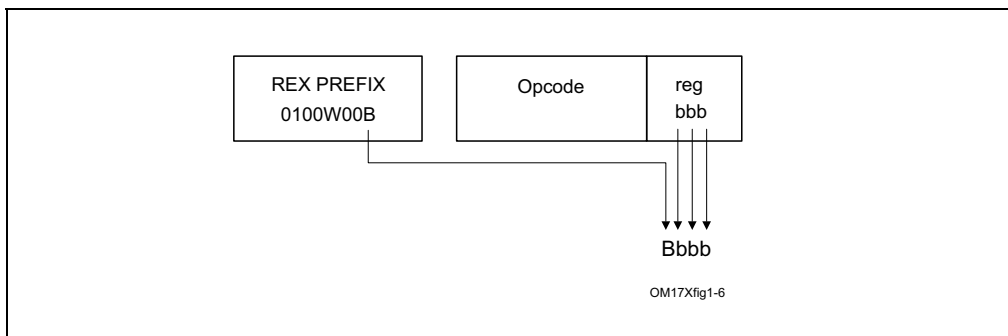


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte’s reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations. Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ≠ 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

NOTES:

* Don't care about value of REX.B

2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8 8877665544332211 MOV RAX,1122334455667788H
```

2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of $\pm 2\text{GB}$ from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

Table 2-7. RIP-Relative Addressing

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	In 64-bit mode, if one wants to use a Disp32 without specifying a base register, one can use a SIB byte encoding (indicated by MODRM.r/m=100) as described in the next row.
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	If mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8-DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

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

2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

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

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.



Figure 2-8. Instruction Encoding Format with VEX Prefix

2.3.2 VEX and the LOCK prefix

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

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

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

2.3.4 VEX and the REX prefix

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

2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

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

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

Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.

- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

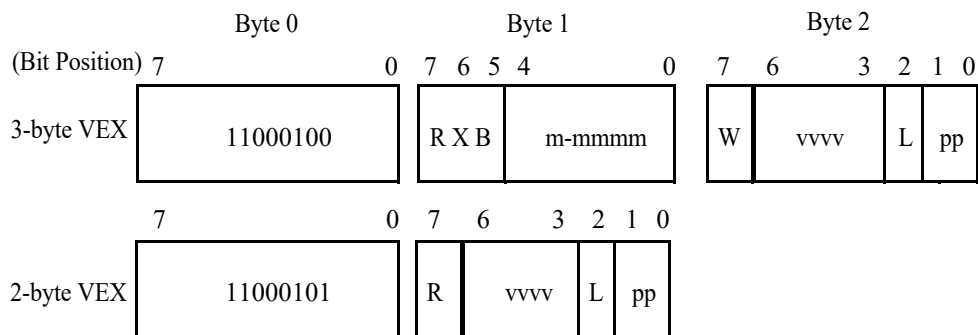
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

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

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

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

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



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

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

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

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

m-mmmm:

00000: Reserved for future use (will #UD)
 00001: implied 0F leading opcode byte
 00010: implied 0F 38 leading opcode bytes
 00011: implied 0F 3A leading opcode bytes
 00100-11111: Reserved for future use (will #UD)

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

L: Vector Length

0: scalar or 128-bit vector
 1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix

00: None
 01: 66
 10: F3
 11: F2

Figure 2-9. VEX bit fields

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

2.3.5.1 VEX Byte 0, bits[7:0]

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

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

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

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

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

This bit is stored in bit inverted format.

2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'

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

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

This bit is stored in bit inverted format.

2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'

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

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

This bit is stored in bit inverted format.

2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'

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

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier

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

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

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

In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

Table 2-8. VEX.vvvv to register name mapping

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

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

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

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

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

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

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv destination

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

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

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

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

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

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

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

2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

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

Table 2-10. VEX.m-mmmm interpretation

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

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

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

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

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

See the following table.

Table 2-11. VEX.L interpretation

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

2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

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

See the following table.

Table 2-12. VEX.pp interpretation

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

2.3.7 The Opcode Byte

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

2.3.8 The MODRM, SIB, and Displacement Bytes

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

2.3.9 The Third Source Operand (Immediate Byte)

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

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

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

2.3.10.1 Vector Length Transition and Programming Considerations

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

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

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

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

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true

if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

2.3.11 AVX Instruction Length

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

2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8D-R15D applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8D-R15D does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte

r32				EAX/ R8D	ECX/ R9D	EDX/ R10D	EBX/ R11D	ESP/ R12D	EBP/ R13D ¹	ESI/ R14D	EDI/ R15D
(In decimal) Base =				0	1	2	3	4	5	6	7
(In binary) Base =				000	001	010	011	100	101	110	111
Scaled Index		SS	Index	Value of SIB Byte (in Hexadecimal)							
VR0/VR8	*1	00	000	00	01	02	03	04	05	06	07
VR1/VR9			001	08	09	0A	0B	0C	0D	0E	0F
VR2/VR10			010	10	11	12	13	14	15	16	17
VR3/VR11			011	18	19	1A	1B	1C	1D	1E	1F
VR4/VR12			100	20	21	22	23	24	25	26	27
VR5/VR13			101	28	29	2A	2B	2C	2D	2E	2F
VR6/VR14			110	30	31	32	33	34	35	36	37
VR7/VR15			111	38	39	3A	3B	3C	3D	3E	3F
VR0/VR8	*2	01	000	40	41	42	43	44	45	46	47
VR1/VR9			001	48	49	4A	4B	4C	4D	4E	4F
VR2/VR10			010	50	51	52	53	54	55	56	57
VR3/VR11			011	58	59	5A	5B	5C	5D	5E	5F
VR4/VR12			100	60	61	62	63	64	65	66	67
VR5/VR13			101	68	69	6A	6B	6C	6D	6E	6F
VR6/VR14			110	70	71	72	73	74	75	76	77
VR7/VR15			111	78	79	7A	7B	7C	7D	7E	7F

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte (Contd.)

r32				EAX/ R8D	ECX/ R9D	EDX/ R10D	EBX/ R11D	ESP/ R12D	EBP/ R13D ¹	ESI/ R14D	EDI/ R15D
(In decimal) Base = (In binary) Base =				0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
Scaled Index		SS	Index	Value of SIB Byte (in Hexadecimal)							
VR0/VR8	*4	10	000	80	81	82	83	84	85	86	87
VR1/VR9			001	88	89	8A	8B	8C	8D	8E	8F
VR2/VR10			010	90	91	92	93	94	95	96	97
VR3/VR11			011	98	99	9A	9B	9C	9D	9E	9F
VR4/VR12			100	A0	A1	A2	A3	A4	A5	A6	A7
VR5/VR13			101	A8	A9	AA	AB	AC	AD	AE	AF
VR6/VR14			110	B0	B1	B2	B3	B4	B5	B6	B7
VR7/VR15			111	B8	B9	BA	BB	BC	BD	BE	BF
VR0/VR8	*8	11	000	C0	C1	C2	C3	C4	C5	C6	C7
VR1/VR9			001	C8	C9	CA	CB	CC	CD	CE	CF
VR2/VR10			010	D0	D1	D2	D3	D4	D5	D6	D7
VR3/VR11			011	D8	D9	DA	DB	DC	DD	DE	DF
VR4/VR12			100	E0	E1	E2	E3	E4	E5	E6	E7
VR5/VR13			101	E8	E9	EA	EB	EC	ED	EE	EF
VR6/VR14			110	F0	F1	F2	F3	F4	F5	F6	F7
VR7/VR15			111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

2.3.12.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

2.4 AVX AND SSE INSTRUCTION EXCEPTION SPECIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

“See Exceptions Type 2”

In this entry, “Type2” can be looked up in Table 2-14.

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

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

NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 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 3B*.

Table 2-14. Exception class description

Exception Class	Instruction set	Mem arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	None
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	None	None
Type 8	AVX	None	None
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.

Table 2-15. Instructions in each Exception Class

Exception Class	Instruction
Type 1	(V)MOVAPD, (V)MOVAPS, (V)MOVVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPs, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, (V)FMADD132PD, (V)FMADD213PD, (V)FMADD231PD, (V)FMADD132PS, (V)FMADD213PS, (V)FMADD231PS, (V)FMADDSUB132PD, (V)FMADDSUB213PD, (V)FMADDSUB231PD, (V)FMADDSUB132PS, (V)FMADDSUB213PS, (V)FMADDSUB231PS, (V)FMSUBADD132PD, (V)FMSUBADD213PD, (V)FMSUBADD231PD, (V)FMSUBADD132PS, (V)FMSUBADD213PS, (V)FMSUBADD231PS, (V)FMSUB132PD, (V)FMSUB213PD, (V)FMSUB231PD, (V)FMSUB132PS, (V)FMSUB213PS, (V)FMSUB231PS, (V)FNMADD132PD, (V)FNMADD213PD, (V)FNMADD231PD, (V)FNMADD132PS, (V)FNMADD213PS, (V)FNMADD231PS, (V)FNMMSUB132PD, (V)FNMMSUB213PD, (V)FNMMSUB231PD, (V)FNMMSUB132PS, (V)FNMMSUB213PS, (V)FNMMSUB231PS, (V)HADDPD, (V)HADDPs, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPD, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDS, (V)ADSS, (V)CMPD, (V)CMPS, (V)COMSD, (V)COMSS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, (V)FMADD132SD, (V)FMADD213SD, (V)FMADD231SD, (V)FMADD132SS, (V)FMADD213SS, (V)FMADD231SS, (V)FMSUB132SD, (V)FMSUB213SD, (V)FMSUB231SD, (V)FMSUB132SS, (V)FMSUB213SS, (V)FMSUB231SS, (V)FNMADD132SD, (V)FNMADD213SD, (V)FNMADD231SD, (V)FNMADD132SS, (V)FNMADD213SS, (V)FNMADD231SS, (V)FNMMSUB132SD, (V)FNMMSUB213SD, (V)FNMMSUB231SD, (V)FNMMSUB132SS, (V)FNMMSUB213SS, (V)FNMMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINS, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMSD, (V)UCOMSS
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, (V)BLENDVPD, (V)BLENDVPS, (V)LDDQU***, (V)MASKMOVDQU, (V)PTEST, (V)PTESTPS, (V)PTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPSEQB, (V)PCMPSEQW, (V)PCMPSEQD, (V)PCMPSEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMAxSB, (V)PMAxSW, (V)PMAxSD, (V)PMAxUB, (V)PMAxUW, (V)PMAxUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUF, (V)PSHUFB, (V)PSHUFD, (V)PSHUFW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS, (V)VBLEND, (V)VPERMD, (V)VPERMPS, (V)VPERMPD, (V)VPERMQ, (V)VPSLLVD, (V)VPSLLVQ, (V)VPSRAVD, (V)VPSRLVD, (V)VPSRLVQ, (V)VPERMILPD, (V)VPERMILPS, (V)VPERM2F128
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)PMOVSXBW, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, (V)LDMXCSR*, (V)VSTMXCSR
Type 6	(V)VEXTRACTF128/VEXTRACTFxxx, (V)VBROADCASTSS, (V)VBROADCASTSD, (V)VBROADCASTF128, (V)VINSERTF128, (V)VMASKMOVPS**, (V)VMASKMOVPD**, (V)VPMASKMOV, (V)VPMASKMOVQ, (V)VPBROADCASTB, (V)VPBROADCASTD, (V)VPBROADCASTW, (V)VPBROADCASTQ, (V)VEXTRACTI128, (V)VINSERTI128, (V)VPERM21128
Type 7	(V)MOVLHPS, (V)MOVHLPs, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMsKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	(V)VZEROALL, (V)VZERoupper
Type 11	(V)VCVTPH2PS, (V)VCVTPS2PH
Type 12	(V)VGATHERDPS, (V)VGATHERDPD, (V)VGATHERQPS, (V)VGATHERQPD, (V)VPGATHERDD, (V)VPGATHERDQ, (V)VPGATHERQD, (V)VPGATHERQQ

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

INSTRUCTION FORMAT

- (**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.
- (***) - PCMPSTRM, PCMPSTRM, PCMPSTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-16. #UD Exception and VEX.W=1 Encoding

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLEND, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		

Table 2-17. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRW, VPMULHW/LW, VPMULLD, VPMULLDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/W/D/DQ, VPUNPCKHQDQ, VPUNPCKLBW/W/D/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	Same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMASKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS
Type 8			
Type 11			
Type 12			

2.4.1 Exceptions Type 1 (Aligned memory reference)

Table 2-18. Type 1 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

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

Table 2-19. Type 2 Class Exception Conditions

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

2.4.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-20. Type 3 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

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

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CRO.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. ¹
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

NOTES:

1. LDDQU, MOVUPD, MOVUPS, PCMPSTRI, PCMPSTRM, PCMPISTRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

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

Table 2-22. Type 5 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

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

Table 2-23. Type 6 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.

2.4.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ≠ 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CRO.TS[bit 3]=1.

2.4.9 Exceptions Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

Table 2-26. Type 11 Class Exception Conditions

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

2.4.10 Exceptions Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm ≠ '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X		For an illegal address in the SS segment.
Stack, #SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
General Protection, #GP(0)				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.

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

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

VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

2.5.1 Exceptions Type 13 (VEX-Encoded GPR Instructions)

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GPR instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

Table 2-28. VEX-Encoded GPR Instructions

Exception Class	Instruction
Type 13	ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

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

Table 2-29. Type 13 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
	X	X	X	X	If VEX.L = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	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. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.6 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

INSTRUCTION FORMAT

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with "load+op" semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality).

2.6.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

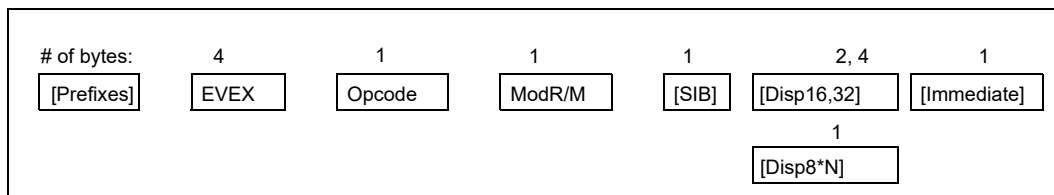


Figure 2-10. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

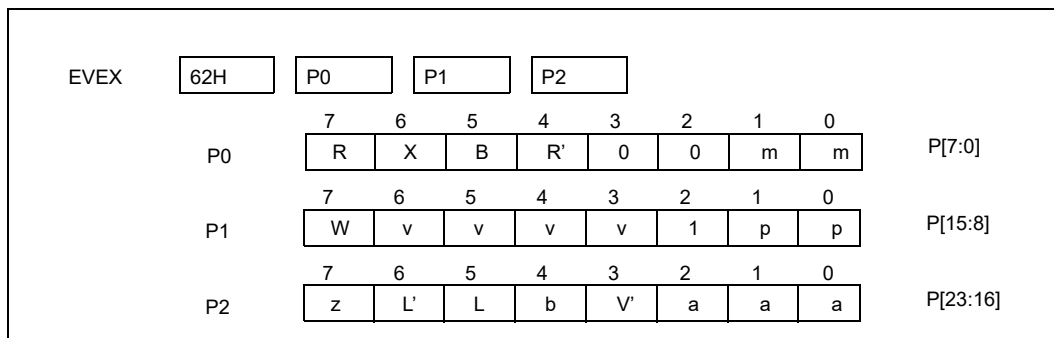


Figure 2-11. Bit Field Layout of the EVEX Prefix¹

NOTES:

1. See Table 2-30 for additional details on bit fields.

Table 2-30. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
--	Reserved	P[3 : 2]	Must be 0.
--	Fixed Value	P[10]	Must be 1.
EVEX.mm	Compressed legacy escape	P[1 : 0]	Identical to low two bits of VEX.mmmmm.
EVEX.pp	Compressed legacy prefix	P[9 : 8]	Identical to VEX.pp.
EVEX.RXB	Next-8 register specifier modifier	P[7 : 5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx). This field is encoded in bit inverted format.
EVEX.R'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg. This bit is stored in inverted format.
EVEX.X	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.vvvv	VVVV register specifier	P[14 : 11]	Same as VEX.vvvv. This field is encoded in bit inverted format.
EVEX.V'	High-16 VVVV/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present. This bit is stored in inverted format.
EVEX.aaa	Embedded opmask register specifier	P[18 : 16]	
EVEX.W	Osize promotion/Opcode extension	P[15]	
EVEX.z	Zeroing/Merging	P[23]	
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22 : 21]	

The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
 - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
 - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).

- Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
- For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.
- Vector length/rounding control specifier: P[22:21] can serve one of three options.
 - Vector length information for packed vector instructions.
 - Ignored for instructions operating on vector register content as a single data element.
 - Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

2.6.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in Section 2.6.3.

Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

	4 ¹	3	[2:0]	Reg. Type	Common Usages
REG	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.V'	EVEX.vvvv		GPR, Vector	2nd Source or Destination
RM	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	0	EVEX.B	modrm.r/m	GPR	memory addressing
INDEX	0	EVEX.X	sib.index	GPR	memory addressing
VIDX	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing

NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-32.

Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode

	[2:0]	Reg. Type	Common Usages
REG	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.vvv	GPR, Vector	2nd Source or Destination
RM	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	modrm.r/m	GPR	Memory Addressing
INDEX	sib.index	GPR	Memory Addressing
VIDX	sib.index	Vector	VSIB Memory Addressing

2.6.3 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using

standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.

- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.6.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

Table 2-33. Opmask Register Specifier Encoding

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
VVVV	VEX.vvvv	k0-k7	2nd Source
RM	modrm.r/m	k0-7	1st Source
{k1}	EVEX.aaa	k0 ¹ -k7	Opmask

NOTES:

1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

2.6.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support “zeroing-masking”.
 - Also allow merging-masking.
- Instructions which require aaa = 000.
 - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking.
 - Require EVEX.z to be set to 0.
 - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
 - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

2.6.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below,

where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.6.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less than dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

Table 2-34. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1tox}	8	8	8	
Half	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1tox}	4	4	4	

Table 2-35. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Mem	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar	8bit	N/A	1	1	1	1 Tuple
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed	32bit	N/A	4	4	4	1 Tuple, memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem	N/A	N/A	8	16	32	SubQword Conversion
Quarter Mem	N/A	N/A	4	8	16	SubDword Conversion
Eighth Mem	N/A	N/A	2	4	8	SubWord Conversion
Mem128	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP	N/A	N/A	8	32	64	VMOVDDUP

2.6.6 EVEX Encoding of Broadcast/Rounding/SAE Support

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with “load+op” semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

2.6.7 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

2.6.8 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.6.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.6.10 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-36.

Table 2-36. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions

Position	P2[4]	P2[6:5]	P2[6:5]
Broadcast/Rounding/SAE Context	EVEX.b	EVEX.L'L	EVEX.RC
Reg-reg, FP Instructions w/ rounding semantic or SAE	Enable static rounding control (SAE implied)	Vector length Implied (512 bit or scalar)	00b: SAE + RNE 01b: SAE + RD 10b: SAE + RU 11b: SAE + RZ
Load+op Instructions w/ memory source	Broadcast Control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Other Instructions (Explicit Load/Store/Broadcast/Gather/Scatter)	Must be 0 (otherwise #UD)		NA

2.6.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

2.6.11.1 State Dependent #UD

In general, attempts to execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

Table 2-37. OS XSAVE Enabling Requirements of Instruction Categories

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	111xxx11b

2.6.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.

Table 2-38. Opcode Independent, State Dependent EVEX Bit Fields

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3 : 2]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[1: 0]	EVEX.mm	if 00b	if 00b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

2.6.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.R	P[7]	ModRM.reg encodes k-reg	if EVEX.R = 0	None (BOUND if EVEX.RX != 11b)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes all other registers	None (valid)	
EVEX.X	P[6]	ModRM.r/m encodes ZMM/YMM/XMM	None (valid)	
		ModRM.r/m encodes k-reg or GPR	None (ignored)	
		ModRM.r/m without SIB/VSIB	None (ignored)	
		ModRM.r/m with SIB/VSIB	None (valid)	
EVEX.B	P[5]	ModRM.r/m encodes k-reg	None (ignored)	None (ignored)
		ModRM.r/m encodes other registers	None (valid)	
		ModRM.r/m base present	None (valid)	
		ModRM.r/m base not present	None (ignored)	
EVEX.R'	P[4]	ModRM.reg encodes k-reg or GPR	if 0	None (ignored)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes ZMM/YMM/XMM	None (valid)	
EVEX.vvvv	P[14 : 11]	vvvv encodes ZMM/YMM/XMM	None (valid)	None (valid) P[14] ignored
		Otherwise	if != 1111b	if != 1111b
EVEX.V'	P[19]	Encodes ZMM/YMM/XMM	None (valid)	None (ignored)
		Otherwise	if 0	None (ignored)

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

Table 2-40. #UD Conditions of Opmask Related Encoding Field

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18 : 16]	Instructions do not use opmask for conditional processing ¹ .	if aaa != 000b	if aaa != 000b
		Opmask used as conditional processing mask and updated at completion ² .	if aaa = 000b	if aaa = 000b;
		Opmask used as conditional processing.	None (valid ³)	None (valid ¹)
EVEX.z	P[23]	Vector instruction using opmask as source or destination ⁴ .	if EVEX.z != 0	if EVEX.z != 0
		Store instructions or gather/scatter instructions.	if EVEX.z != 0	if EVEX.z != 0
		Instruction supporting conditional processing mask with EVEX.aaa = 000b.	if EVEX.z != 0	if EVEX.z != 0
VEX.vvvv	Varies	K-regs are instruction operands not mask control.	if vvvv = 0xxx	None

NOTES:

1. E.g., VPBROADCASTMxxx, VPMOVM2x, VPMOVx2M.

2. E.g., Gather/Scatter family.

3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in KO.

4. E.g., VFPClassPD/PS, VCMPB/D/Q/W family, VPMOVM2x, VPMOVx2M.

Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

Table 2-41. #UD Conditions Dependent on EVEX.b Context

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid ¹)	None (valid ¹)
		Other reg-reg, FP instructions that can cause #XM.	None (valid ²)	None (valid ²)
		Other reg-mem instructions in Table 2-34.	None (valid ³)	None (valid ³)
		Other instruction classes ⁴ in Table 2-35.	If EVEX.b > 0	If EVEX.b > 0

NOTES:

1. L'L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L'L specifies vector length, see Table 2-36, supports {sae} syntax.
3. L'L specifies vector length, see Table 2-36, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

2.6.12 Device Not Available

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

2.6.13 Scalar Instructions

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

2.7 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

Table 2-42. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	Explicitly aligned, w/ fault suppression	None
Type E1NF	Vector Non-temporal Stores	Explicitly aligned, no fault suppression	None
Type E2	FP Vector Load+op	Support fault suppression	Yes
Type E2NF	FP Vector Load+op	No fault suppression	Yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	Yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	Yes
Type E4	Integer Vector Load+op	Support fault suppression	No
Type E4NF	Integer Vector Load+op	No fault suppression	No
Type E5	Legacy-like Promotion	Varies, Support fault suppression	No

Table 2-42. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	No
Type E6	Post AVX Promotion	Varies, w/ fault suppression	No
Type E6NF	Post AVX Promotion	Varies, no fault suppression	No
Type E7NM	Register-to-register op	None	None
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	None
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	None
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	None
Type E11	VCVTPH2PS, VCVTPS2PH	Half Vector Length, w/ fault suppression	Yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	None
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-43. EVEX Instructions in each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPs, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2DQ, VCVTPS2UDQs, VCVTQ2PD, VCVTQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VDIVPD, VDIVPS, VEXP2PD, VEXP2PS, VFIXUPIMMPD, VFIXUPIMMPS, VFMADDxxxPD, VFMADDxxxPS, VFMADDSUBxxxPD, VFMADDSUBxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VSCALEFPD, VSCALEFPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS
Type E3	VADDSd, VADDSs, VCMPSD, VCMPSs, VCVTPS2QQ, VCVTPS2UQQ, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VCVTTPS2QQ, VCVTTPS2UQQ, VDIVSD, VDIVSS, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS, VSQRSD, VSQRSS, VSUBSD, VSUBSS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSD2USI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTSS2USI, VCVTTSD2SI, VCVTTSD2USI, VCVTTSS2SI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS, VUCOMISD, VUCOMISS
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VBLENDMPD, VBLENDMPS, VFPCLASSPD, VFPCLASSPS, VORPD, VORPS, VPABSD, VPABSQ, VPADD, VPADDQ, VPAND, VPANDQ, VPANDND, VPANDNQ, VPBLENDMB, VPBLENDMD, VPBLENDMQ, VPBLENDMw, VPCMPD, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPMADD52LUQ, VPMADD52HUQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPROLD, VPROLQ, VPROLVD, VPROLVQ, VPRORD, VPRORQ, VPRORVD, VPRORVQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRAVw, VPSRAVD, VPSRAVw, VPSRAVQ, VPSRLD, VPSRLQ) ¹ , VPSUBD, VPSUBQ, VPSUBUSB, VPSUBUSw, VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VXORPD, VXORPS

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
E4.nb ²	VCOMPRESSPD, VCOMPRESSPS, VEXPANDPD, VEXPANDPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VMOVUPD, VMOVUPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPB, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPCMPW, VPCMPUB, VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRW, VPMULHUW, VPMULHW, VPMULLW, VPSLLVW, VPSLLW, VPSRAW, VPSRLVW, VPSRLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW
Type E4NF	VALIGND, VALIGNQ, VPACKSSDW, VPACKUSDW, VPCONFLICTD, VPCONFLICTQ, VPERMD, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VPMULTISHIFTQB, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VSHUFFD, VSHUFFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS
E4NF.nb ²	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) ³ , VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W
Type E5	PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBW, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ, VCVTDQ2PD, VCVTUDQ2PD, VPMOVsx _{xx} , VPMOVz _{xx}
Type E5NF	VMOVDDUP
Type E6	VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VBROADCASTSD, VBROADCASTSS, VFPCCLASSSD, VFPCCLASSSS, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSD, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E6NF	VEXTRACTF32X4, VEXTRACTF32X8, VEXTRACTF64X2, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI32X8, VEXTRACTI64X2, VEXTRACTI64X4, VINSERTF32X4, VINSERTF32X8, VINSERTF64X2, VINSERTF64X4, VINSERTI32X4, VINSERTI32X8, VINSERTI64X2, VINSERTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D
Type E7NM.128 ⁴	VMOVHLP, VMOVLHPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW) ⁵ , VPMOVb2M, VPMOVD2M, VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVQ2M, VPMOVW2M
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS
Type E10NF	(VCVTISI2SD, VCVTUSI2SD) ⁶
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPFODPD, VGATHERPFODPS, VGATHERPFOQPD, VGATHERPFOQPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPFODPD, VSCATTERPFODPS, VSCATTERPFOQPD, VSCATTERPFOQPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

NOTES:

1. Operand encoding Full tupletype with immediate.
2. Embedded broadcast is not supported with the “.nb” suffix.
3. Operand encoding Mem128 tupletype.

4. #UD raised if EVEX.L'L != 00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

2.7.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

Table 2-44. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

Table 2-45. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.7.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

Table 2-46. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> State requirement, Table 2-37 not met. Opcode independent #UD condition in Table 2-38. Operand encoding #UD conditions in Table 2-39. Opmask encoding #UD condition of Table 2-40. If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access 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, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.7.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

Table 2-47. Type E3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access 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, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

Table 2-48. Type E3NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access 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, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.7.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

Table 2-49. Type E4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 2-43). ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

Table 2-50. Type E4NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43). ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.7.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

Table 2-51. Type E5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 2-52. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If 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		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.7.6 Exceptions Type E6 and E6NF

Table 2-53. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
			X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

Table 2-54. Type E6NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.7.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

Table 2-55. Type E7NM Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.7.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

Table 2-56. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 00b (VL=128).
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

Table 2-57. Type E9NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 00b (VL=128).
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If 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		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.7.9 Exceptions Type E10 and E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding and do not cause no SIMD FP exception, support memory fault suppression follow exception class E10.

Table 2-58. Type E10 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E10NF.

Table 2-59. Type E10NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.7.10 Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

Table 2-60. Type E11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

2.7.11 Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions)

Table 2-61. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512). ▪ If vvvv != 1111b.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
X	X	X	X	If index = destination register (gather operation).	
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

Table 2-62. Type E12NP Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

2.8 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below.

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

Table 2-63. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

Exception conditions of Opmask instructions that address memory are listed as Type K21.

Table 2-64. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	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. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

7. Updates to Chapter 3, Volume 2A

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

Changes to this chapter include typo corrections and updates to various instructions. Updates to the following instructions: CALL, CLRSSBSY, COMISD, COMISS, CPUID, CVTDQ2PD, CVTPD2PI, ENDBR32, ENDBR64, FRSTOR, IN, INT n/INTO/INT3/INT1, IRET/IRETD/IRETQ, JMP, and LFENCE.

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	Z0	V/V	NA	Complement carry flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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.
- **NF_x** — Indicates the use of 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

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
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											
R8B	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9B	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10B	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11B	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12B	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13B	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14B	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15B	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.[128,256].[66,F2,F3].OF/OF3A/OF38.[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:

- **128,256:** VEX.L field can be 0 (denoted by VEX.128 or VEX.LZ) or 1 (denoted by VEX.256). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:
 - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L = 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
 - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception (e.g. VMOVLPS).
 - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.
 - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.
- **66,F2,F3:** The presence or absence of these values map to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the "opcode" byte in the

same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.

- **0F,0F3A,0F38:** The presence maps to a valid encoding of the VEX.mmmmm field. Only three encoded values of VEX.mmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmm field must be encoded using the 3-byte form of VEX prefix.
- **0F,0F3A,0F38 and 2-byte/3-byte VEX:** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.
- **W0:** VEX.W=0.
- **W1:** VEX.W=1.
- The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.
- **WIG:** can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.
- If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.
- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in 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 of 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.[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:

- **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.

NOTE

Previously, the terms NDS, NDD and DDS were used in instructions with an EVEX (or VEX) prefix. These terms indicated that the vvvv field was valid for encoding, and specified register usage. These terms are no longer necessary and are redundant with the instruction operand encoding tables provided with each instruction. The instruction operand encoding tables give explicit details on all operands, indicating where every operand is stored and if they are read or written. If vvvv is not listed as an operand in the instruction operand encoding table, then EVEX (or VEX) vvvv must be 0b1111.

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 (R8B - R15B) 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 R8B - R15B 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.
- **m80bcd** — A Binary Coded Decimal (BCD) operand in memory, 80 bits.
- **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 i^{th} element from the top of the FPU register stack ($i := 0$ 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 tuple types. The tuple type for an instruction is listed in the operand encoding definition table where applicable.

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.
- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

MAXVL Definition

XCR0 Component	MAXVL
XCR0.SSE	128
XCR0.AVX	256
XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}	512

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.

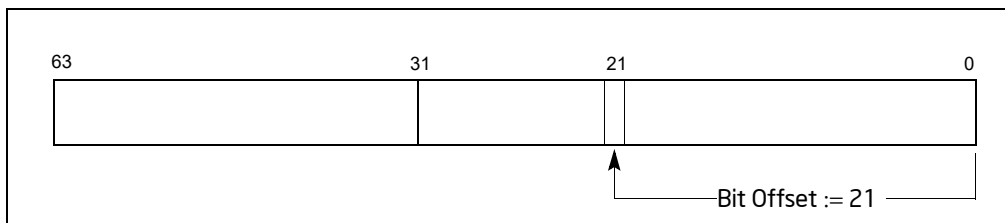


Figure 3-1. Bit Offset for `BIT[RAX, 21]`

If BitBase is a memory address, the BitOffset 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).

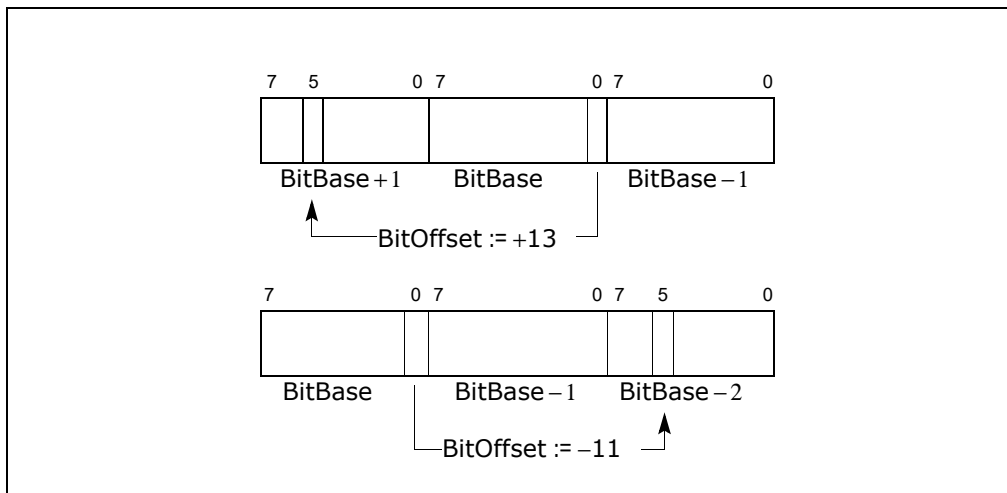


Figure 3-2. Memory Bit Indexing

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`, `_mm256_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 <code>stdint.h</code>)
<code>__int64</code>	<code>int64_t</code>
<code>unsigned __int64</code>	<code>uint64_t</code>
<code>__int32</code>	<code>int32_t</code>
<code>unsigned __int32</code>	<code>uint32_t</code>
<code>__int16</code>	<code>int16_t</code>
<code>unsigned __int16</code>	<code>uint16_t</code>

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	INT3 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.

CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 <i>cw</i>	CALL <i>rel16</i>	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 <i>cd</i>	CALL <i>rel32</i>	D	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 <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF /2	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF /2	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A <i>cd</i>	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A <i>cp</i>	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> . 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 <i>m16:32</i>	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 <i>m16:64</i>	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 (<i>r</i>)	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 procedure's 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 a 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 is 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.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

Instruction ordering. Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Certain situations may lead to the next sequential instruction after a near indirect CALL being speculatively executed. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

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);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush8B(RIP);
          FI;
          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);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush4B(EIP);
          FI;
          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);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
            ShadowStackPush4B(IP);
          FI;
          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);
    IF ShadowStackEnabled(CPL)
      ShadowStackPush8B(RIP);
    FI;
    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);
    IF ShadowStackEnabled(CPL)
      ShadowStackPush4B(EIP);
    FI;
    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);
    IF ShadowStackEnabled(CPL)
      (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
      ShadowStackPush4B(IP);
    FI;
    EIP := tempEIP;
  FI;
FI;rel/abs
IF (Call near indirect, absolute indirect)
  IF EndbranchEnabledAndNotSuppressed(CPL)
    IF CPL = 3
      THEN
        IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
          THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
          FI;
        ELSE
          IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
            THEN
              IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            FI;
          FI;
        FI;
      FI;
    FI;
  FI;
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); 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); FI;

IF DPL > CPL

THEN #GP(new code segment); FI;

IF segment not present

THEN #NP(new code segment); FI;


```

IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN
        tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_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 ShadowStackEnabled(CPL)
    IF OperandSize = 32
        THEN
            tempPushLIP = CSBASE + EIP;
        ELSE
            IF OperandSize = 16
                THEN
                    tempPushLIP = CSBASE + IP;
                ELSE (* OperandSize = 64 *)
                    tempPushLIP = RIP;
            FI;
        FI;
    tempPushCS = CS;
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;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0

```

```

    (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
    IF (SSP & 0xFFFFFFFF00000000 != 0)
        THEN #GP(0); FI;
    FI;
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFFFFFFFFF8H
    ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(tempPushLIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
            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 target mode = Compatibility mode
        THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (IA32_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 ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA & CS.L
            tempPushLIP = RIP
        ELSE
            tempPushLIP = CSBASE + EIP;
        FI;
        tempPushCS = CS;
    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;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
        FI;
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFFFFFFFFF8H
    ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
    ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
    FI;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    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;
    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;
IF ShadowStackEnabled(CPL) AND CPL = 3
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
            ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                IA32_PL3_SSP := LA_adjust(SSP);
        FI;
    FI;
CPL := CodeSegment(DPL)
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := IA32_PLi_SSP; (* where i is the CPL *)
    IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
        THEN #GP(0); FI;

```

```

IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;
IF oldSS.DPL != 3
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(oldSSP);
FI;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;
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
    IF ShadowStackEnabled(CPL)
        (* Align to next 8 byte boundary *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (SSP - 4)
        SSP = SSP & 0xFFFFFFFFFFFFFFF8H;
        (* push cs:rip:ssp on shadow stack *)

```

```

ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
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 descriptor is not a TSS segment
  THEN #GP(TSS selector); FI;
IF TSS descriptor specifies that the TSS is busy
  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)	<p>If the target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the gate is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If “supervisor Shadow Stack” token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.</p> <p>If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
#GP(selector)	<p>If a code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>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.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>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.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>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.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	<p>If a code segment, data segment, call gate, task gate, or TSS is not present.</p>
#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is NULL.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p> <p>If the new stack segment is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>

#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. If target mode is compatibility mode and SSP is not in low 4GB. If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned. If "supervisor Shadow Stack" token on new shadow stack is marked busy. If destination mode is 32-bit mode or compatibility mode, but SSP address in "super-visor shadow" stack token is beyond 4GB. If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#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.

CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 CLRSSBSY m64	M	V/V	CET_SS	Clear busy flag in supervisor shadow stack token reference by m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
M	NA	ModRM:r/m (r, w)	NA	NA	NA

Description

Clear busy flag in supervisor shadow stack token reference by m64. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

Operation

```
IF (CR4.CET = 0)
  THEN #UD; FI;
```

```
IF (IA32_S_CET.SH_STK_EN = 0)
  THEN #UD; FI;
```

```
IF CPL > 0
  THEN GP(0); FI;
```

```
SSP_LA = Linear_Address(mem operand)
```

```
IF SSP_LA not aligned to 8 bytes
```

```
  THEN #GP(0); FI;
```

```
expected_token_value = SSP | BUSY_BIT      (* busy bit - bit position 0 - must be set *)
```

```
new_token_value = SSP                      (* Clear the busy bit *)
```

```
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
```

```
  invalid_token := 1; FI
```

```
(* Set the CF if invalid token was detected *)
```

```
RFLAGS.CF = (invalid_token == 1) ? 1 : 0;
```

```
RFLAGS.ZF,PF,AF,OF,SF := 0;
```

```
SSP := 0
```

Flags Affected

CF is set if an invalid token was detected, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If destination is located in a non-writeable segment. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. If CPL is not 0.
#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	The CLRSSBSY instruction is not recognized in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD	Same exceptions as in protected mode.
#GP(0)	Same exceptions as in protected mode.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes. If CPL is not 0. If the memory address is in a non-canonical form. If token is invalid.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	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 [operation](#) 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 OF 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	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 operation 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	Z0	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
Z0	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 some Intel processors, 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. *)2
CPUID.EAX = 1FH (* Returns V2 Extended Topology Enumeration leaf. *)2
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.
2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
<i>Basic CPUID Information</i>		
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
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. ** The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.
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.		
<i>Deterministic Cache Parameters Leaf</i>		
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 227.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>Bits 07 - 05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache.</p> <p>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**, ****, *****.</p> <p>EBX Bits 11 - 00: L = System Coherency Line Size**. Bits 21 - 12: P = Physical Line partitions**. Bits 31 - 22: W = Ways of associativity**.</p> <p>ECX Bits 31-00: S = Number of Sets**.</p> <p>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.</p> <p>Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels.</p> <p>Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits.</p> <p>Bits 31 - 03: Reserved = 0.</p> <p>NOTES:</p> <p>* 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.</p> <p>** Add one to the return value to get the result.</p> <p>***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.</p> <p>**** 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.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>
	<i>MONITOR/MWAIT Leaf</i>
05H	<p>EAX Bits 15 - 00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.</p> <p>EBX Bits 15 - 00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.</p> <p>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.</p>

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.
<i>Thermal and Power Management Leaf</i>		
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. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bit 15: HWP Capabilities. Highest Performance change is supported if set. Bit 16: HWP PECL override is supported if set. Bit 17: Flexible HWP is supported if set. Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set. Bit 19: HW_FEEDBACK. IA32_HW_FEEDBACK_PTR MSR, IA32_HW_FEEDBACK_CONFIG MSR, IA32_PACKAGE_THERM_STATUS MSR bit 26, and IA32_PACKAGE_THERM_INTERRUPT MSR bit 25 are supported if set. Bit 20: Ignoring Idle Logical Processor HWP request is supported if set. Bits 31 - 21: 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 (1BOH). Bits 31 - 04: Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>EDX</p> <p>Bits 7-0: Bitmap of supported hardware feedback interface capabilities. 0 = When set to 1, indicates support for performance capability reporting. 1 = When set to 1, indicates support for energy efficiency capability reporting. 2-7 = Reserved</p> <p>Bits 11-8: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result.</p> <p>Bits 31-16: Index (starting at 0) of this logical processor’s row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure.</p> <p>NOTE: Bits 0 and 1 will always be set together.</p>
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>	
07H	<p style="text-align: center;">Sub-leaf 0 (Input ECX = 0). *</p> <p>EAX</p> <p>Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.</p> <p>EBX</p> <p>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. Bit 16: AVX512F. Bit 17: AVX512DQ. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bit 21: AVX512_IFMA. Bit 22: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bit 26: AVX512PF. (Intel® Xeon Phi™ only.) Bit 27: AVX512ER. (Intel® Xeon Phi™ only.) Bit 28: AVX512CD. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bit 30: AVX512BW. Bit 31: AVX512VL.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
ECX	<p>Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.)</p> <p>Bit 01: AVX512_VBMI.</p> <p>Bit 02: UMIP. Supports user-mode instruction prevention if 1.</p> <p>Bit 03: PKU. Supports protection keys for user-mode pages if 1.</p> <p>Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions).</p> <p>Bit 05: WAITPKG.</p> <p>Bit 06: AVX512_VBMI2.</p> <p>Bit 07: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SPP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PLO_SSP.</p> <p>Bit 08: GFNI.</p> <p>Bit 09: VAES.</p> <p>Bit 10: VPCLMULQDQ.</p> <p>Bit 11: AVX512_VNNI.</p> <p>Bit 12: AVX512_BITALG.</p> <p>Bits 13: Reserved.</p> <p>Bit 14: AVX512_VPOPCNTDQ. (Intel® Xeon Phi™ only.)</p> <p>Bits 16 - 15: Reserved.</p> <p>Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.</p> <p>Bit 22: RDPID and IA32_TSC_AUX are available if 1.</p> <p>Bits 24 - 23: Reserved.</p> <p>Bit 25: CLDEMOTE. Supports cache line demote if 1.</p> <p>Bit 26: Reserved.</p> <p>Bit 27: MOVDIRI. Supports MOVDIRI if 1.</p> <p>Bit 28: MOVDIR64B. Supports MOVDIR64B if 1.</p> <p>Bit 29: Reserved.</p> <p>Bit 30: SGX_LC. Supports SGX Launch Configuration if 1.</p> <p>Bit 31: PKS. Supports protection keys for supervisor-mode pages if 1.</p>
EDX	<p>Bit 01: Reserved.</p> <p>Bit 02: AVX512_4VNNIW. (Intel® Xeon Phi™ only.)</p> <p>Bit 03: AVX512_4FMAPS. (Intel® Xeon Phi™ only.)</p> <p>Bit 04: Fast Short REP MOV.</p> <p>Bits 09-05: Reserved.</p> <p>Bit 10: MD_CLEAR supported.</p> <p>Bits 14-11: Reserved.</p> <p>Bit 15: Hybrid. If 1, the processor is identified as a hybrid part.</p> <p>Bits 19-16: Reserved.</p> <p>Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs.</p> <p>Bits 25 - 21: Reserved.</p> <p>Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB).</p> <p>Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP).</p> <p>Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH).</p> <p>Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	<p>Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR.</p> <p>IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. A bit being set in this MSR indicates that a model specific feature is supported; software must still consult CPUID family/model/stepping to determine the behavior of the enumerated feature as features enumerated in IA32_CORE_CAPABILITIES may have different behavior on different processor models.</p> <p>Additionally, on hybrid parts (CPUID.07H.0H:EDX[15]=1), software must consult the native model ID and core type from the Hybrid Information Enumeration Leaf.</p> <p>Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD).</p> <p>NOTE:</p> <p>* 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.</p>	
<i>Direct Cache Access Information Leaf</i>		
09H	EAX EBX ECX EDX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H). Reserved. Reserved. Reserved.
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX EBX ECX EDX	<p>Bits 07 - 00: Version ID of architectural performance monitoring.</p> <p>Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor.</p> <p>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter.</p> <p>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.</p> <p>Bit 00: Core cycle event not available if 1.</p> <p>Bit 01: Instruction retired event not available if 1.</p> <p>Bit 02: Reference cycles event not available if 1.</p> <p>Bit 03: Last-level cache reference event not available if 1.</p> <p>Bit 04: Last-level cache misses event not available if 1.</p> <p>Bit 05: Branch instruction retired event not available if 1.</p> <p>Bit 06: Branch mispredict retired event not available if 1.</p> <p>Bits 31 - 07: Reserved = 0.</p> <p>Reserved = 0.</p> <p>Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1).</p> <p>Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1).</p> <p>Bits 14 - 13: Reserved = 0.</p> <p>Bit 15: AnyThread deprecation.</p> <p>Bits 31 - 16: Reserved = 0.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>Extended Topology Enumeration Leaf</i>	
OBH	<p>NOTES:</p> <p><i>CPUID leaf 1FH is a preferred superset to leaf OBH. Intel recommends first checking for the existence of Leaf 1FH before using leaf OBH.</i></p> <p>Most of Leaf OBH output depends on the initial value in ECX.</p> <p>The EDX output of leaf OBH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order.</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0.</p> <p>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].</p> <p>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.</p> <p>EBX Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31 - 16: Reserved.</p> <p>ECX Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.</p> <p>EDX Bits 31 - 00: x2APIC ID the current logical processor.</p> <p>NOTES:</p> <p>* Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p> <p>** 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.</p> <p>*** 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.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>		
0DH		<p>NOTES: Leaf 0DH main leaf (ECX = 0).</p>
	EAX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCRO. XCRO[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 12 - 10: Reserved. Bit 13: Used for IA32_XSS. Bits 15 - 14: Reserved. Bit 16: Used for IA32_XSS. Bits 31 - 17: Reserved.
	EBX	Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. 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 XCRO.
	EDX	Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCRO. XCRO[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>		
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 XSAVEC/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 XCRO. Bit 08: PT state. Bit 09: Used for XCRO. Bit 10: Reserved. Bit 11: CET user state. Bit 12: CET supervisor state. Bit 13: HDC state. Bits 15 - 14: Reserved. Bit 16: HWP state. Bits 31 - 17: 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
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)</i>	
0DH	<p>NOTES:</p> <p>Leaf 0DH output depends on the initial value in ECX.</p> <p>Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR.</p> <p>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n ($0 \leq n \leq 31$) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n ($32 \leq n \leq 63$) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].</p> <p>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, <i>n</i>.</p> <p>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, <i>n</i>, does not map to a valid bit in the XCRO register*.</p> <p>ECX Bit 00 is set if the bit <i>n</i> (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit <i>n</i> is instead supported in XCRO. 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, <i>n</i>, is invalid*.</p> <p>EDX This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*; otherwise it is reserved.</p>
<i>Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>	
0FH	<p>NOTES:</p> <p>Leaf 0FH output depends on the initial value in ECX.</p> <p>Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.</p> <p>EAX Reserved.</p> <p>EBX Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.</p> <p>ECX Reserved.</p> <p>EDX Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>	
0FH	<p>NOTES: Leaf 0FH output depends on the initial value in ECX.</p> <p>EAX Reserved.</p> <p>EBX Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.</p> <p>ECX Maximum range (zero-based) of RMID of this resource type.</p> <p>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.</p>
<i>Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>	
10H	<p>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.</p> <p>EAX Reserved.</p> <p>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.</p> <p>ECX Reserved.</p> <p>EDX Reserved.</p>
<i>L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>	
10H	<p>NOTES: Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.</p> <p>EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.</p> <p>ECX Bits 01 - 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved.</p> <p>EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.</p>
<i>L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 2)</i>	
10H	<p>NOTES: Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.</p> <p>EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.</p> <p>ECX Bits 31 - 00: Reserved.</p> <p>EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)</i>	
10H	<p>NOTES: Leaf 10H output depends on the initial value in ECX.</p> <p>EAX Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved.</p> <p>EBX Bits 31 - 00: Reserved.</p> <p>ECX Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.</p> <p>EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.</p>
<i>Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)</i>	
12H	<p>NOTES: Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H);EBX[SGX] = 1.</p> <p>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. Bits 04 - 02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVRTCHILD, EDECVIRTCHILD, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bits 31 - 07: Reserved.</p> <p>EBX Bits 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.</p> <p>ECX Bits 31 - 00: Reserved.</p> <p>EDX Bits 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is $2^{(EDX[7:0])}$. Bits 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is $2^{(EDX[15:8])}$. Bits 31 - 16: Reserved.</p>
<i>Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)</i>	
12H	<p>NOTES: Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H);EBX[SGX] = 1.</p> <p>EAX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.</p> <p>EBX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.</p> <p>ECX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.</p> <p>EDX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.</p>
<i>Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)</i>	
12H	<p>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.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>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.</p> <p>Type 0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.</p> <p>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.</p> <p>EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved.</p> <p>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.</p> <p>EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.</p>
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>	
14H	<p>NOTES: Leaf 14H main leaf (ECX = 0).</p> <p>EAX Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</p> <p>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.</p> <p>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 MaskOffsetTableOffset 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.</p> <p>EDX Bits 31 - 00: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)</i>		
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.
	EDX	Bits 31 - 00: Reserved.
<i>Time Stamp Counter and Nominal Core Crystal Clock Information Leaf</i>		
15H	<p>NOTES:</p> <p>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.</p>	
	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.
<i>Processor Frequency Information Leaf</i>		
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.
	<p>NOTES:</p> <p>* 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.</p> <p>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.</p>	
<i>System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)</i>		
17H	<p>NOTES:</p> <p>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.</p>	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	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.
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)</i>		
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.	
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)</i>		
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.
<i>Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)</i>		
18H	NOTES: Each sub-leaf enumerates a different address translation structure. 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. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the <i>Intel® 64 and IA-32 Architectures Optimization Reference Manual</i> for details of a particular product. ** Add one to the return value to get the result.	
	EAX	Bits 31 - 00: Reports the maximum input value of supported sub-leaf in leaf 18H.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>EBX Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.</p> <p>ECX Bits 31 - 00: S = Number of Sets.</p> <p>EDX Bits 04 - 00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB*. 00100b: Load Only TLB. Hit on loads; fills on both loads and stores. 00101b: Store Only TLB. Hit on stores; fill on stores. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.</p>
<i>Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)</i>	
18H	<p>NOTES:</p> <p>Each sub-leaf enumerates a different address translation structure. 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. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.</p> <p>* Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the <i>Intel® 64 and IA-32 Architectures Optimization Reference Manual</i> for details of a particular product.</p> <p>** Add one to the return value to get the result.</p> <p>EAX Bits 31 - 00: Reserved.</p> <p>EBX Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.</p> <p>ECX Bits 31 - 00: S = Number of Sets.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 04 - 00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB*. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.
<i>Hybrid Information Enumeration Leaf (EAX = 1AH, ECX = 0)</i>		
1AH	EAX	Enumerates the native model ID and core type. Bits 31-24: Core type 10H: Reserved 20H: Intel Atom® 30H: Reserved 40H: Intel® Core™ Bits 23-0: Native model ID of the core. The core-type and native mode ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>V2 Extended Topology Enumeration Leaf</i>		
1FH		<p>NOTES:</p> <p><i>CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH and using this if available.</i></p> <p>Most of Leaf 1FH output depends on the initial value in ECX.</p> <p>The EDX output of leaf 1FH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order.</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0.</p> <p>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].</p> <p>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.</p> <p>EBX Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.</p> <p>ECX Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.</p> <p>EDX Bits 31- 00: x2APIC ID the current logical processor.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	<p>NOTES:</p> <p>* Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p> <p>** 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.</p> <p>*** 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: Module. 4: Tile. 5: Die. 6-255: Reserved.</p>	
<i>Unimplemented CPUID Leaf Functions</i>		
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.	
<i>Extended Function CPUID Information</i>		
80000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information. Reserved. Reserved. Reserved.
80000001H	EAX EBX ECX EDX	<p>Extended Processor Signature and Feature Bits.</p> <p>Reserved.</p> <p>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.</p> <p>Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET.** 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.</p> <p>NOTES:</p> <p>* LAHF and SAHF are always available in other modes, regardless of the enumeration of this feature flag. ** Intel processors support SYSCALL and SYSRET only in 64-bit mode. This feature flag is always enumerated as 0 outside 64-bit mode.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
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 ECX EDX	Reserved = 0. Reserved = 0. 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. Reserved = 0. NOTES: * L2 associativity field encodings: 00H - Disabled 01H - 1 way (direct mapped) 02H - 2 ways 03H - Reserved 04H - 4 ways 05H - Reserved 06H - 8 ways 07H - See CPUID leaf 04H, sub-leaf 2** 08H - 16 ways 09H - Reserved 0AH - 32 ways 0BH - 48 ways 0CH - 64 ways 0DH - 96 ways 0EH - 128 ways 0FH - Fully associative ** CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2
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.
80000008H	EAX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0.
	<p>NOTES:</p> <p>* If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.</p>	

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 (* "inel", 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.

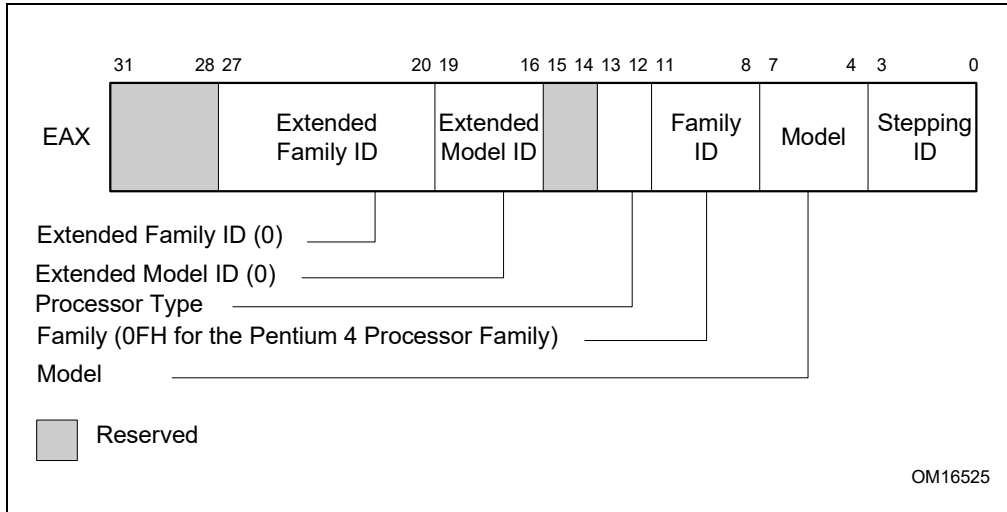


Figure 3-6. Version Information Returned by CPUID in EAX

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 20 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.

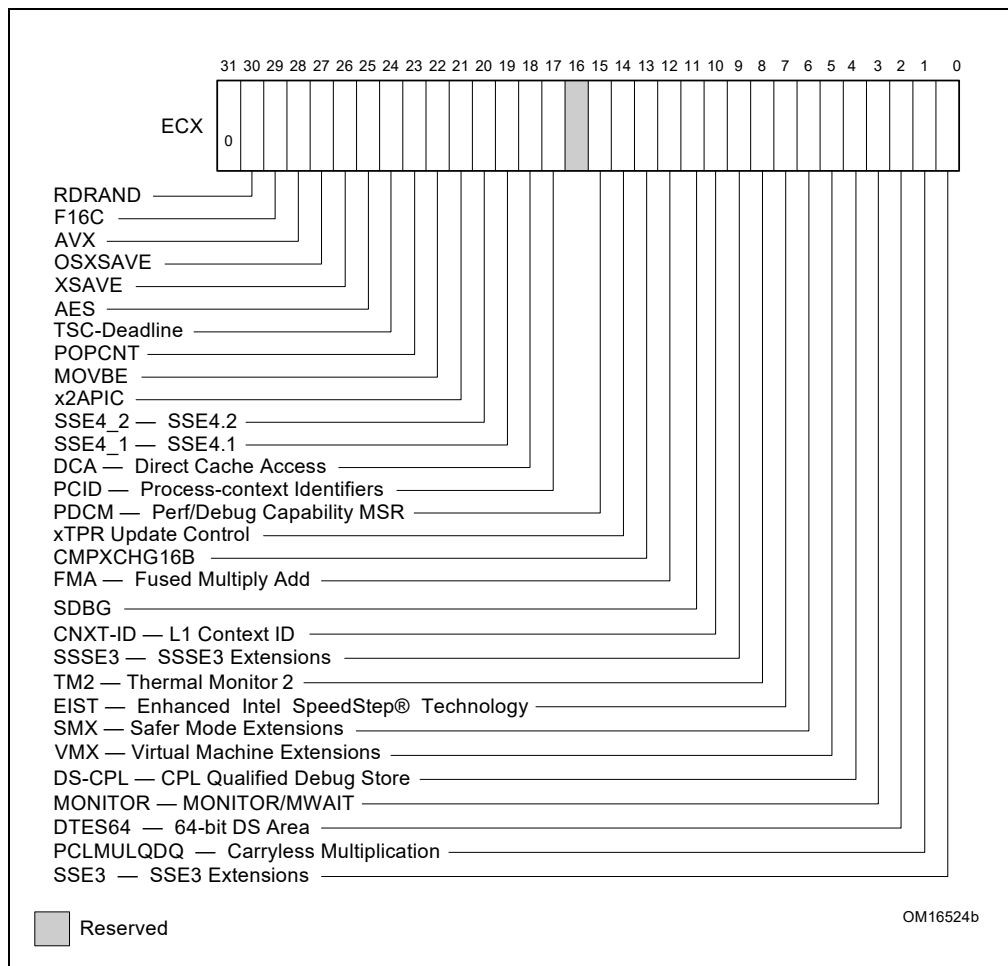


Figure 3-7. Feature Information Returned in the ECX Register

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 XCRO.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCRO 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.

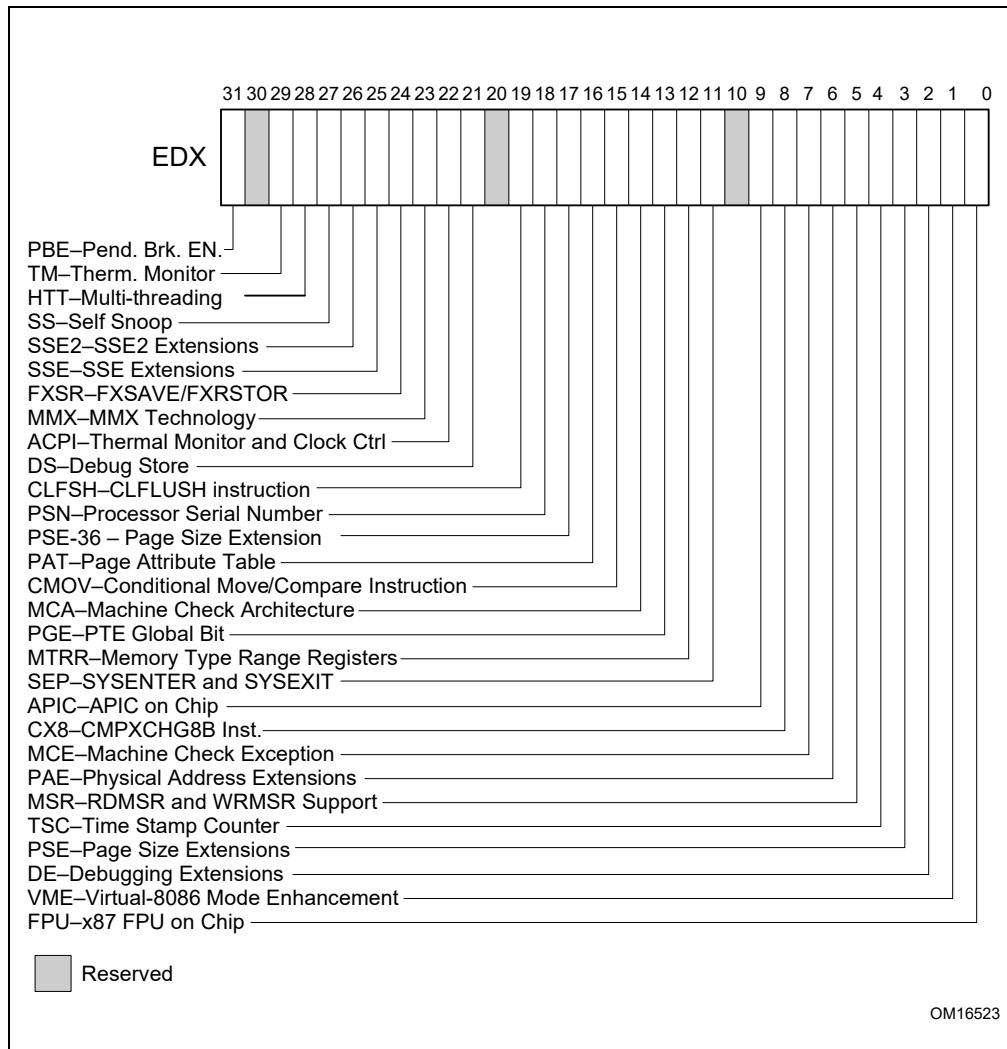
**Figure 3-8. Feature Information Returned in the EDX Register**

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 processor event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C</i>).
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.

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- μ op, 8-way set associative
71H	Cache	Trace cache: 16 K- μ op, 8-way set associative
72H	Cache	Trace cache: 32 K- μ op, 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 1GByte 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
FOH	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
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- μ op, 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

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

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

CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.

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 $\geq 0BH$, 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 ($n \geq 1$, 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.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

INPUT EAX = 1AH: Returns Hybrid Information

When CPUID executes with EAX set to 1AH, the processor returns information about hybrid capabilities. See Table 3-8.

INPUT EAX = 1FH: Returns V2 Extended Topology Information

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is >= 1FH, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. 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 20 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.

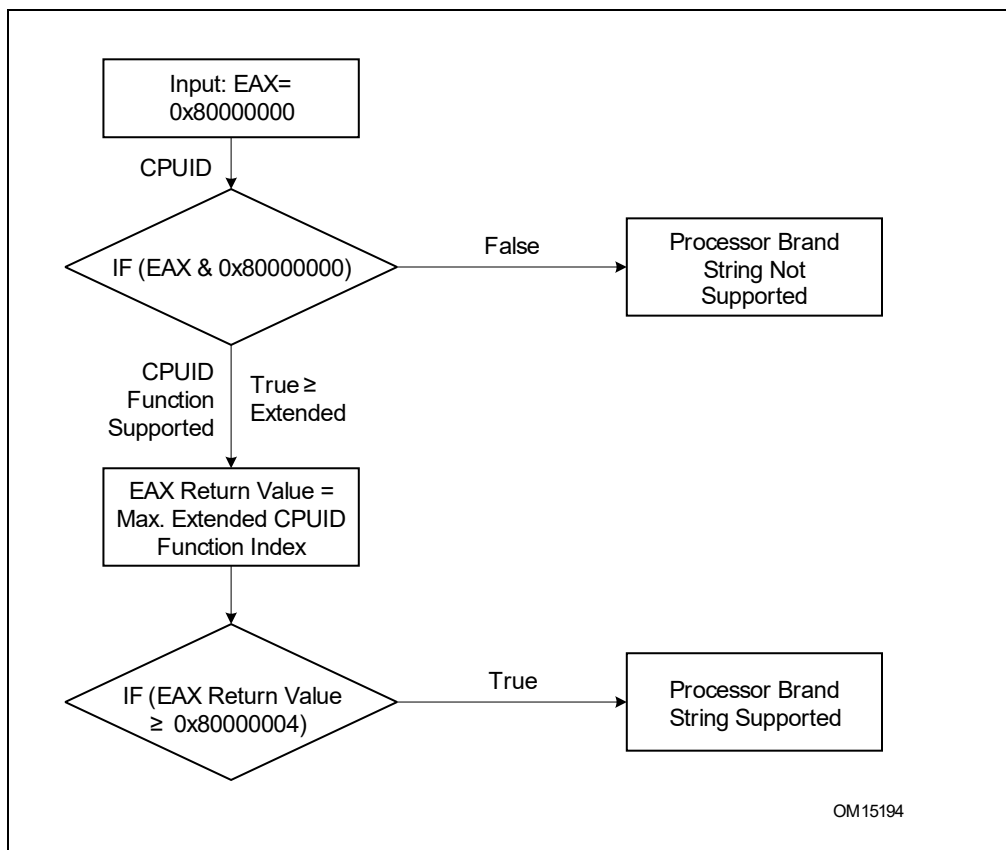


Figure 3-9. Determination of Support for the Processor Brand String

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	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P)R" "itne" "R(mu"
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4)" " UPC" "0051" "\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.

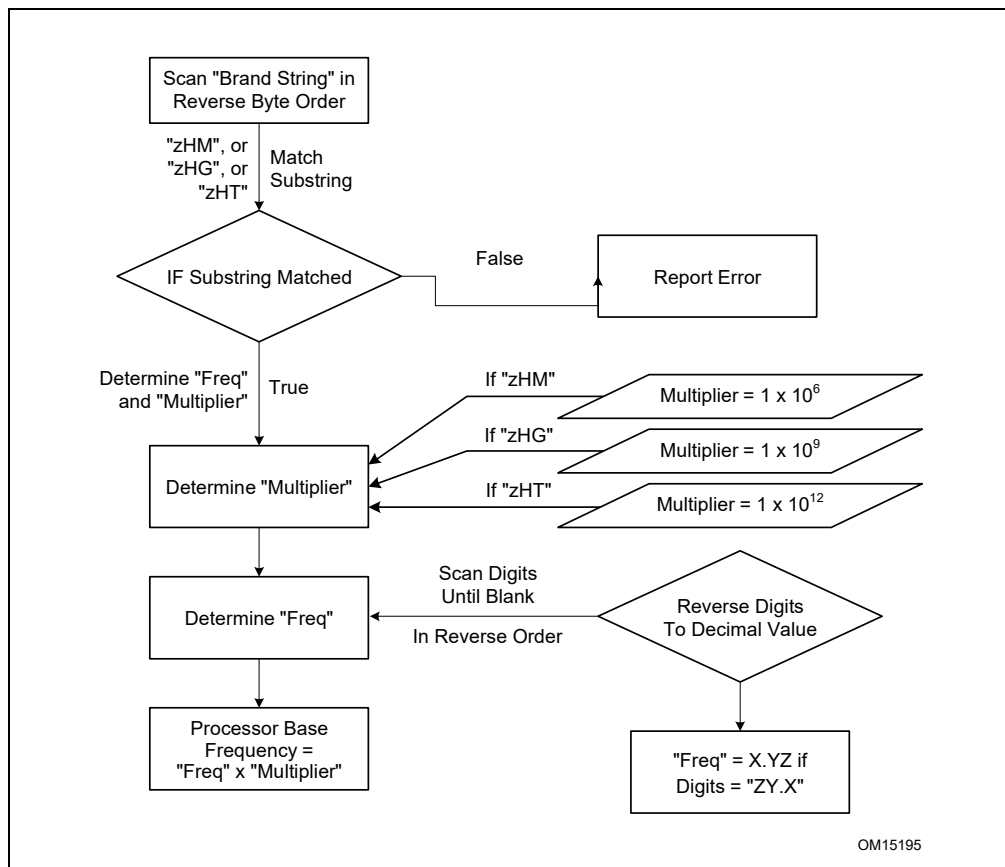


Figure 3-10. Algorithm for Extracting Processor Frequency

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 = 18H:

EAX := Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-8. *)
 EBX := Deterministic Address Translation Parameters Enumeration Leaf;
 ECX := Deterministic Address Translation Parameters Enumeration Leaf;
 EDX := Deterministic Address Translation Parameters Enumeration Leaf;

BREAK;

EAX = 1AH:

EAX := Hybrid Information Enumeration Leaf; (* See Table 3-8. *)
 EBX := Hybrid Information Enumeration Leaf;
 ECX := Hybrid Information Enumeration Leaf;
 EDX := Hybrid Information Enumeration Leaf;

BREAK;

EAX = 1FH:
 EAX := V2 Extended Topology Enumeration Leaf; (* See Table 3-8. *)
 EBX := V2 Extended Topology Enumeration Leaf;
 ECX := V2 Extended Topology Enumeration Leaf;
 EDX := V2 Extended Topology 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	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m64/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	B	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	B	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half	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 (MAXVL-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 (MAXVL-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.

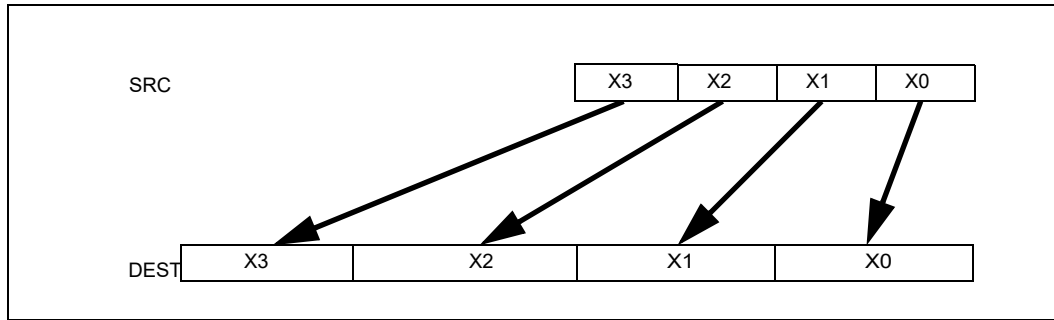


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2D /r CVTPD2PI <i>mm, xmm/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

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

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FB ENDBR32	Z0	V/V	CET_IBT	Terminate indirect branch in 32 bit and compatibility mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA	NA

Description

Terminate an indirect branch in 32 bit and compatibility mode.

Operation

```
IF EndbranchEnabled(CPL) & (IA32_EFER.LMA = 0 | (IA32_EFER.LMA=1 & CS.L = 0))
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = IDLE
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = IDLE
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
```

Flags Affected

None.

Exceptions

None.

ENDBR64—Terminate an Indirect Branch in 64-bit Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FA ENDBR64	Z0	V/V	CET_IBT	Terminate indirect branch in 64 bit mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA	NA

Description

Terminate an indirect branch in 64 bit mode.

Operation

IF EndbranchEnabled(CPL) & IA32_EFER.LMA = 1 & CS.L = 1

IF CPL = 3

THEN

IA32_U_CET.TRACKER = IDLE

IA32_U_CET.SUPPRESS = 0

ELSE

IA32_S_CET.TRACKER = IDLE

IA32_S_CET.SUPPRESS = 0

FI;

FI;

Flags Affected

None.

Exceptions

None.

FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD /4	FRSTOR <i>m94/108byte</i>	Valid	Valid	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

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

Operation

```
FPUControlWord := SRC[FPUControlWord];
FPUStatusWord := SRC[FPUStatusWord];
FPUTagWord := SRC[FPUTagWord];
FPUDataPointer := SRC[FPUDataPointer];
FPUInstructionPointer := SRC[FPUInstructionPointer];
FPULastInstructionOpcode := SRC[FPULastInstructionOpcode];
```

```
ST(0) := SRC[ST(0)];
ST(1) := SRC[ST(1)];
ST(2) := SRC[ST(2)];
ST(3) := SRC[ST(3)];
ST(4) := SRC[ST(4)];
ST(5) := SRC[ST(5)];
ST(6) := SRC[ST(6)];
ST(7) := SRC[ST(7)];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.

IN—Input from Port

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	I	Valid	Valid	Input byte from <i>imm8</i> I/O port address into AL.
E5 <i>ib</i>	IN AX, <i>imm8</i>	I	Valid	Valid	Input word from <i>imm8</i> I/O port address into AX.
E5 <i>ib</i>	IN EAX, <i>imm8</i>	I	Valid	Valid	Input dword from <i>imm8</i> I/O port address into EAX.
EC	IN AL,DX	ZO	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	ZO	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	ZO	Valid	Valid	Input doubleword from I/O port in DX into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	<i>imm8</i>	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 19, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST := SRC; (* Read from selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST := SRC; (* Read from selected I/O port *)
```

FI;

Flags Affected

None

Protected Mode Exceptions

- #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

INT *n*/INT0/INT3/INT1—Call to Interrupt Procedure

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT3	Z0	Valid	Valid	Generate breakpoint trap.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INT0	Z0	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	Z0	Valid	Valid	Generate debug trap.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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 INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.¹ Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. 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, INT3, and INT1 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.

Each of the INT *n*, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.

even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug.)

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 3-52. 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.
- S/W Applies to INT n, INT3, and INTO, but not to INT1.

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.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

Instruction ordering. Instructions following an INT n may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the INT n have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible). This applies also to the INTO, INT3, and INT1 instructions, but not to executions of INTO when EFLAGS.OF = 0.

Operation

The following operational description applies not only to the INT n , INTO, INT3, or INT1 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 $(\text{num} \& \text{FCH}) \mid \text{ext}$; (2) if `idt` is 1, the error code is $(\text{num} \ll 3) \mid 2 \mid \text{ext}$.

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 (INT n , INT3, or INTO), `EXT` is 0; otherwise (including INT1), `EXT` is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT  $n$ )
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT  $n$  *)
      ELSE
        IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT  $n$ )
          THEN
            Consult bit  $n$  of the software interrupt redirection bit map in the TSS;
            IF bit  $n$  is clear
              THEN (* redirect interrupt to 8086 program interrupt handler *)
                Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                Push CS;
                Push IP;
                IF IOPL = 3
                  THEN IF := 0; (* Clear interrupt flag *)
                  ELSE VIF := 0; (* Clear virtual interrupt flag *)
                FI;
                TF := 0; (* Clear trap flag *)
                load CS and EIP (lower 16 bits only) from entry  $n$  in interrupt vector table referenced from TSS;
              ELSE
                IF IOPL = 3
                  THEN GOTO PROTECTED-MODE;
                  ELSE #GP(0); (* Bit 0 of error code is 0 because INT  $n$  *)
                FI;
            FI;
          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;
      FI;
    FI;
  FI;

```

```

        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; does not apply to INT1 *)
        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, INT3, or INTO; does not apply to INT1 *)
        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;

```

```

        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(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(error_code(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 *)
            NewSSP := IA32_Pli_SSP (* where i = new code-segment DPL *)
        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 *)
IF IDT-gate IST = 0
    THEN
        NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
    ELSE
        NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST << 3)
        (* Check if shadow stacks are enabled at CPL 0 *)
        IF ShadowStackEnabled(CPL 0)
            THEN NewSSP := 8 bytes loaded from NewSSPAddress; FI;
    FI;
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 & FFFFFFFF0H;
                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;
IF ShadowStackEnabled(CPL) AND CPL = 3
THEN
    IF IA32_EFER.LMA = 0
    THEN IA32_PL3_SSP := SSP;
    ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
        IA32_PL3_SSP := LA_adjust(SSP);
    FI;
FI;
CPL := new code-segment DPL;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
oldSSP := SSP
SSP := NewSSP
IF SSP & 0x07 != 0
    THEN #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)

```

```

IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;

    IF oldSS.DPL != 3
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(oldSSP);
    FI;
FI;
IF EndbranchEnabled (CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
    IA32_S_CET.SUPPRESS = 0
FI;
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 *)
        NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
        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 := Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) := 0;
CPL := 0;
IF IDT gate is 32-bit
    THEN
        EIP := Gate(instruction pointer);
    ELSE (* IDT gate is 16-bit *)
        EIP := Gate(instruction pointer) AND 0000FFFFH;
FI;
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := NewSSP
    IF SSP & 0x07 != 0
        THEN #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT (* Set the busy bit *)

```

```

IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
  THEN #GP(0); FI;
  IF oldSS.DPL != 3
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(oldSSP);
  FI;
FI;
IF EndbranchEnabled (CPL)
  IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
  IA32_S_CET.SUPPRESS = 0
FI;
(* Start execution of new routine in Protected Mode *)
END;

```

INTRA-PRIVILEGE-LEVEL-INTERRUPT:

```

NewSSP = SSP;
CHECK_SS_TOKEN = 0
(* 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;
  IF ShadowStackEnabled(CPL)
    THEN
      NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT_gate IST << 3)
      NewSSP := 8 bytes loaded from NewSSPAddress
      CHECK_SS_TOKEN = 1
    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 *)
          FI;
        FI;
      FI;
    FI;
  FI;

```

```

ELSE
    IF instruction pointer from IDT gate contains a non-canonical address
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    RSP := NewRSP & FFFFFFFF0H;
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 ShadowStackEnabled(CPL)
        IF CHECK_SS_TOKEN == 1
            THEN
                IF NewSSP & 0x07 != 0
                    THEN #GP(0); FI;
                IF ((IA32_EFER.LMA and CS.L) = 0 AND NewSSP[63:32] != 0)
                    THEN #GP(0); FI;
                expected_token_value = NewSSP (* busy bit - bit position 0 - must be clear *)
                new_token_value = NewSSP | BUSY_BIT (* Set the busy bit *)
                IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value, expected_token_value) != expected_token_value
                    THEN #GP(0); FI;
            FI;
        (* Align to next 8 byte boundary *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (NewSSP - 4)
        SSP = newSSP & 0xFFFFFFFFF8H;
        (* push cs:rip:ssp on shadow stack *)
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled (CPL)
        IF CPL = 3

```

```

    THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
    ELSE
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;
FI;
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)	<p>If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt, trap, or task gate is NULL.</p> <p>If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt, trap, or task gate.</p> <p>If an interrupt is generated by the INT <i>n</i>, INT3, or INTO instruction and the DPL of an interrupt, trap, or task gate is less than the CPL.</p> <p>If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p> <p>If SSP in IA32_PLi_SSP (where <i>i</i> is the new CPL) is not 8 byte aligned.</p> <p>If “supervisor Shadow Stack” token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.</p> <p>If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where <i>i</i> is the new CPL).</p>
#SS(error_code)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>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.</p>
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	<p>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.</p> <p>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.</p>

	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 <i>n</i> , INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate 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 gate, trap gate, 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 gate. If an interrupt is generated by INT <i>n</i> , INT3, or INTO and the DPL of an interrupt, trap, or task gate 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 gate, trap gate, 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.
#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)	<p>If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.</p> <p>If the segment selector in the 64-bit interrupt or trap gate is NULL.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If the vector points to a gate which is in non-canonical space.</p> <p>If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.</p> <p>If the descriptor pointed to by the gate selector is outside the descriptor table limit.</p> <p>If the descriptor pointed to by the gate selector is in non-canonical space.</p> <p>If the descriptor pointed to by the gate selector is not a code segment.</p> <p>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.</p> <p>If the descriptor pointed to by the gate selector has $DPL > CPL$.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If "supervisor shadow stack" token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB.</p> <p>If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
#SS(error_code)	<p>If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.</p> <p>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).</p>
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	<p>If an attempt to load RSP from the TSS causes an access to non-canonical space.</p> <p>If the RSP from the TSS is outside descriptor table limits.</p>
#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/IRETQ—Interrupt Return

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CF	IRET	Z0	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	Z0	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	Z0	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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 performs 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.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for CET details.

Instruction ordering. IRET is a serializing instruction. See Section 8.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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 *)
        FI;
      FI;
    FI;
  
```

```

        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;
    FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    (* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
    IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
        THEN #GP(0); FI;
    shadowStackEnabled = ShadowStackEnabled(CPL)
    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 *)
    IF shadowStackEnabled
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;

```

```

CPL := 3;
(* Resume execution in Virtual-8086 mode *)
tempOldSSP = SSP;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF shadowStackEnabled
    expected_token_value = tempOldSSP | BUSY_BIT    (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                    (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
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;
            IF ShadowStackEnabled(CPL)
                (* check if 8 byte aligned *)
                IF SSP AND 0x7 ≠ 0

```

```

    THEN #CP(FAR-RET/IRET); FI;
IF CS(RPL) != 3
    THEN
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* Do 64 bit compare to detect bits beyond 15 being set *)
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
        IF CSBASE + RIP != tempSsEIP
            THEN #CP(FAR-RET/IRET); FI;
        (* check if 4 byte aligned *)
        IF tempSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
tempOldCPL = CPL;
CPL := CS(RPL);
IF OperandSize = 64
    THEN
        RSP := tempRSP;
        SS := tempSS;
    ELSE
        ESP := tempESP;
        SS := tempSS;
    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;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
    ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP;
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;

```

```

FOR each SegReg in (ES, FS, GS, and DS)
  DO
    tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
    IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
      THEN (* Segment register invalid *)
        SegmentSelector := 0; (*Segment selector becomes 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;
      IF ShadowStackEnabled(CPL)
        IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
          THEN #CP(FAR-RET/IRET); FI;
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
          THEN #CP(FAR-RET/IRET); FI;
        IF CSBASE + RIP != tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
          THEN #CP(FAR-RET/IRET); FI;
        IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
          THEN #CP(FAR-RET/IRET); FI;
        IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
          ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
          THEN #GP(0); FI;
        FI;
      IF ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA = 1
          (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
            through an IDT with a non-zero IST *)
          (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies if the
            stack switch was to self stack and if so, do not try to free the token on this shadow stack. If the
  
```



```

tempSSP was not to same stack then there was a stack switch so do attempt to free the token *)
  IF tempSSP != SSP
    THEN
      expected_token_value = SSP | BUSY_BIT      (* busy bit - bit position 0 - must be set *)
      new_token_value = SSP                      (* clear the busy bit *)
      shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
    FI;
  FI;
  SSP := tempSSP
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;
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();
            FI;
          FI;
        FI;
      FI;
    FI;
  FI;

```

```

                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. 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 reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP (Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned. If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB. If return instruction pointer from stack and shadow stack do not match.

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)	<p>If EFLAGS.NT[bit 14] = 1.</p> <p>If the return code segment selector is NULL.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the return instruction pointer is non-canonical.</p>
#GP(Selector)	<p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p> <p>If the return stack segment is not present.</p>
#NP (selector)	If the return code 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.
#CP (Far-RET/IRET)	<p>If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.</p> <p>If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.</p> <p>If return instruction pointer from stack and shadow stack do not match.</p>

JMP—Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB <i>cb</i>	JMP <i>rel8</i>	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 <i>cw</i>	JMP <i>rel16</i>	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 <i>cd</i>	JMP <i>rel32</i>	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF <i>14</i>	JMP <i>r/m16</i>	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended <i>r/m16</i> . Not supported in 64-bit mode.
FF <i>14</i>	JMP <i>r/m32</i>	M	N.S.	Valid	Jump near, absolute indirect, address given in <i>r/m32</i> . Not supported in 64-bit mode.
FF <i>14</i>	JMP <i>r/m64</i>	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-bit offset from register or memory
EA <i>cd</i>	JMP <i>ptr16:16</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
EA <i>cp</i>	JMP <i>ptr16:32</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
FF <i>15</i>	JMP <i>m16:16</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:16</i>
FF <i>15</i>	JMP <i>m16:32</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:32</i> .
REX.W FF <i>15</i>	JMP <i>m16:64</i>	D	Valid	N.E.	Jump far, absolute indirect, address given in <i>m16:64</i> .

Instruction Operand Encoding

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

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to -128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is 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). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP 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.

A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here 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 address 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 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 Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

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 jump to be performed.

If the selected descriptor is for a code segment, a far jump 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 jump 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, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform 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 jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

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

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for CET details.

In 64-Bit Mode. The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

Instruction ordering. Instructions following a far jump may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far jump have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Certain situations may lead to the next sequential instruction after a near indirect JMP being speculatively executed. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect JMP in order to block speculative execution.

Operation

```

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP := RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP := DEST;
        FI;
      ELSE
        IF near relative jump
          THEN
            tempEIP := EIP + DEST; (* EIP is instruction following JMP instruction*)
          ELSE (* Near absolute jump *)
            tempEIP := DEST;
          FI;
        FI;
      IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
        and tempEIP outside code segment limit
          THEN #GP(0); FI
      IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
      FI;
      IF OperandSize = 32
        THEN
          EIP := tempEIP;
        ELSE
          IF OperandSize = 16
            THEN (* OperandSize = 16 *)
              EIP := tempEIP AND 0000FFFFH;
            ELSE (* OperandSize = 64)

```

```

        RIP := tempRIP;
    FI;
FI;
IF (JMP near indirect, absolute indirect)
    IF EndbranchEnabledAndNotSuppressed(CPL)
        IF CPL = 3
            THEN
                IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI;
                    FI;
                FI;
            FI;
        FI;
    FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP := DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS := DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP := tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP := tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
            FI;
        FI;
    FI;
IF far jump and (PE = 1 and VM = 0)
    (* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
            or segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (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
                    call gate
                    THEN #GP(segment selector); FI;
            FI;
        Depending on type and access rights:
        GO TO CONFORMING-CODE-SEGMENT;

```

```

        GO TO NONCONFORMING-CODE-SEGMENT;
        GO TO CALL-GATE;
        GO TO TASK-GATE;
        GO TO TASK-STATE-SEGMENT;
ELSE
    #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF DPL > CPL
    THEN #GP(segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP := DEST(Offset);
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH;
FI;
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
tempEIP outside code segment limit
    THEN #GP(0); FI
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
    FI;
FI;
CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL
EIP := tempEIP;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    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(code segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP := DEST(Offset);
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI;

```



```

IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
    THEN #GP(0); FI
IF tempEIP is non-canonical THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
        FI;
    FI;
    CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) := CPL;
    EIP := tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
            FI;
        FI;
    FI;
END;

```

CALL-GATE:

```

IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); FI;
IF call gate not present
    THEN #NP(call gate selector); FI;
IF call gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); FI;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
or code-segment segment descriptor is conforming and DPL > CPL
or code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); FI;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
or code-segment segment descriptor has both L-Bit and D-bit set)
    THEN #GP(code segment selector); FI;
IF code segment is not present
    THEN #NP(code-segment selector); FI;
tempEIP := DEST(Offset);
IF GateSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
    THEN #GP(0); FI
CS := DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;

```

```

EIP := tempEIP;
IF EndbranchEnabled(CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
END;
TASK-GATE:
  IF task gate DPL < CPL
  or task gate DPL < task gate segment-selector RPL
    THEN #GP(task gate selector); FI;
  IF task gate not present
    THEN #NP(gate selector); FI;
  Read the TSS segment selector in the task-gate descriptor;
  IF TSS segment selector local/global bit is set to local
  or index not within GDT limits
  or descriptor is not a TSS segment
  or TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); FI;
  IF TSS not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
  IF TSS DPL < CPL
  or TSS DPL < TSS segment-selector RPL
  or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); FI;
  IF TSS is not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;

```

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)	<p>If offset in target operand, call gate, or TSS is beyond the code segment limits.</p> <p>If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p>
#GP(selector)	If the segment 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

(When not using a call gate.) If 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 selector in a call gate does not indicate it is a code segment.

If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.

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 a memory operand effective address is outside the SS segment limit.

#NP (selector) If the code segment being accessed is not present.

If call gate, task gate, or TSS 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. (Only occurs when fetching target from memory.)

#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 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 the target operand is beyond the code segment limits.

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. (Only occurs when fetching target from memory.)

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

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

If target offset in destination operand is non-canonical.

If 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 64-bit gate is NULL.

If transitioning to compatibility mode and the SSP is beyond 4GB.

#GP(selector) If the code segment or 64-bit call gate is outside descriptor table limits.

If the code segment or 64-bit call gate overlaps non-canonical space.

If the segment descriptor from a 64-bit call gate is in 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, 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 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.

If the code segment is non-conforming and $CPL \neq DPL$.

If the code segment is confirming and $CPL < DPL$.

#NP(selector) If a code segment or 64-bit call gate is not present.

#UD (64-bit mode only) If a far jump 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.

LFENCE—Load Fence

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE E8 LFENCE	Z0	V/V	SSE2	Serializes load operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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 (even speculatively) 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.

8. Updates to Chapter 4, Volume 2B

Change bars and green text show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U*.

Changes to this chapter include updates to various instructions. Updates to the following instructions: MAXPD, MAXPS, MAXSD, MAXSS, MFENCE, MINPD, MINPS, MINSR, MINSS, MOV, MOVBE, MOVDIRI/MOVDIR64B, MOVDQA, VMOVDQA32/64, MOVNTI, MOVQ2DQ, MOVSB/MOVSDB, PACKSSWB/PACKSSDW, POPF/POPFD/POPFQ, PSADBW, PUSHF/PUSHFD/PUSHFQ, RET, RSTORSSP, SAVEPREVSSP, SQRSSQ, SYSCALL, SYSENTER, SYSEXIT, UCOMISD, and UCOMISS.

MAXPD—Maximum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, 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 MAXPD 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 64-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 (MAXVL-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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
  ELSE DEST := SRC2;
  FI;
}
```

VMAXPD (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] := MAX(SRC1[i+63:i], SRC2[63:0])
        ELSE
          DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] := 0 ; zeroing-masking
        FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMAXPD (VEX.256 encoded version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] := MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] := MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] := 0
```

VMAXPD (VEX.128 encoded version)

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


MAXPD (128-bit Legacy SSE version)

DEST[63:0] := MAX(DEST[63:0], SRC[63:0])

DEST[127:64] := MAX(DEST[127:64], SRC[127:64])

DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m512d __mm512_max_pd(__m512d a, __m512d b);

VMAXPD __m512d __mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b,);

VMAXPD __m512d __mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);

VMAXPD __m512d __mm512_max_round_pd(__m512d a, __m512d b, int);

VMAXPD __m512d __mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);

VMAXPD __m512d __mm512_maskz_max_round_pd(__mmask8 k, __m512d a, __m512d b, int);

VMAXPD __m256d __mm256_mask_max_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);

VMAXPD __m256d __mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b);

VMAXPD __m128d __mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);

VMAXPD __m128d __mm_maskz_max_pd(__mmask8 k, __m128d a, __m128d b);

VMAXPD __m256d __mm256_max_pd(__m256d a, __m256d b);

(V)MAXPD __m128d __mm_max_pd(__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

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	A	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	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.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	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.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	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 (MAXVL-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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) 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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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	A	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) 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[MAXVL-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[MAXVL-1:128] := 0

```

MAXSD (128-bit Legacy SSE version)

```

DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-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	A	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	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 (MAXVL: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 (MAXVL: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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) 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[MAXVL-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[MAXVL-1:128] := 0

```

MAXSS (128-bit Legacy SSE version)

```

DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-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	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE F0 MFENCE	Z0	V/V	SSE2	Serializes load and store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, 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 MINPD 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 64-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 (MAXVL-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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
  ELSE DEST := SRC2;
  FI;
}
```

VMINPD (EVEX encoded version)

(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] := MIN(SRC1[i+63:i], SRC2[63:0])

ELSE

DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE DEST[i+63:i] := 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VMINPD (VEX.256 encoded version)

DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])

DEST[191:128] := MIN(SRC1[191:128], SRC2[191:128])

DEST[255:192] := MIN(SRC1[255:192], SRC2[255:192])

VMINPD (VEX.128 encoded version)

DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])

DEST[MAXVL-1:128] := 0

MINPD (128-bit Legacy SSE version)

DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])

DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```

VMINPD __m512d __mm512_min_pd( __m512d a, __m512d b);
VMINPD __m512d __mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_min_round_pd( __m512d a, __m512d b, int);
VMINPD __m512d __mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d __mm512_maskz_min_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d __mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d __mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);
VMINPD __m128d __mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d __mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);
VMINPD __m256d __mm256_min_pd ( __m256d a, __m256d b);
MINPD __m128d __mm_min_pd ( __m128d a, __m128d b);

```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.128.OF.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.OF.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.OF.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	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.256.OF.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	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.512.OF.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	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 (MAXVL-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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) 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[MAXVL-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[MAXVL-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[MAXVL-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, __m256 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	A	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	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 (MAXVL-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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
  ELSE DEST := SRC2;
  FI;
}

```

MINSND (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[MAXVL-1:128] := 0

```

MINSND (VEX.128 encoded version)

```

DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

MINSND (128-bit Legacy SSE version)

```

DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VMINSND __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSND __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSND __m128d _mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSND __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	A	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.LIG.F3.0F.WO 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	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 MINSR 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 (MAXVL: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 (MAXVL-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 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) 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[MAXVL-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[MAXVL-1:128] := 0

```

MINSS (128-bit Legacy SSE version)

```

DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-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.

MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 20/r MOV r32, CR0–CR7	MR	N.E.	Valid	Move control register to r32.
OF 20/r MOV r64, CR0–CR7	MR	Valid	N.E.	Move extended control register to r64.
REX.R + OF 20 /0 MOV r64, CR8	MR	Valid	N.E.	Move extended CR8 to r64. ¹
OF 22 /r MOV CR0–CR7, r32	RM	N.E.	Valid	Move r32 to control register.
OF 22 /r MOV CR0–CR7, r64	RM	Valid	N.E.	Move r64 to extended control register.
REX.R + OF 22 /0 MOV CR8, r64	RM	Valid	N.E.	Move r64 to extended CR8. ¹

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *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
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8–R15). Use of the REX.W prefix or 66H prefix is ignored. Use of the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

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

DEST := SRC;

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write 1 to CR4.PCIDE.</p> <p>If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

Real-Address Mode Exceptions

#GP	<p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write 1 to CR4.PCIDE.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.
--------	---

Compatibility Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.</p> <p>If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.</p> <p>If an attempt is made to write a 1 to any reserved bit in CR3.</p> <p>If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.</p>

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
--------	--

If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).

If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.

If an attempt is made to clear CR0.PG[bit 31].

If an attempt is made to write a 1 to any reserved bit in CR4.

If an attempt is made to write a 1 to any reserved bit in CR8.

If an attempt is made to write a 1 to any reserved bit in CR3.

If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

#UD

If the LOCK prefix is used.

If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

If the REX.R prefix is used to specify a register other than CR8.

MOVBE—Move Data After Swapping Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 F0 /r MOVBE r16, m16	RM	V/V	MOVBE	Reverse byte order in <i>m16</i> and move to <i>r16</i> .
OF 38 F0 /r MOVBE r32, m32	RM	V/V	MOVBE	Reverse byte order in <i>m32</i> and move to <i>r32</i> .
REX.W + OF 38 F0 /r MOVBE r64, m64	RM	V/N.E.	MOVBE	Reverse byte order in <i>m64</i> and move to <i>r64</i> .
OF 38 F1 /r MOVBE m16, r16	MR	V/V	MOVBE	Reverse byte order in <i>r16</i> and move to <i>m16</i> .
OF 38 F1 /r MOVBE m32, r32	MR	V/V	MOVBE	Reverse byte order in <i>r32</i> and move to <i>m32</i> .
REX.W + OF 38 F1 /r MOVBE m64, r64	MR	V/N.E.	MOVBE	Reverse byte order in <i>r64</i> and move to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA
MR	ModRM:r/m (<i>w</i>)	ModRM:reg (<i>r</i>)	NA	NA

Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

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

TEMP := SRC

```

IF ( OperandSize = 16)
  THEN
    DEST[7:0] := TEMP[15:8];
    DEST[15:8] := TEMP[7:0];
  ELES IF ( OperandSize = 32)
    DEST[7:0] := TEMP[31:24];
    DEST[15:8] := TEMP[23:16];
    DEST[23:16] := TEMP[15:8];
    DEST[31:23] := TEMP[7:0];
  ELSE IF ( OperandSize = 64)
    DEST[7:0] := TEMP[63:56];
    DEST[15:8] := TEMP[55:48];
    DEST[23:16] := TEMP[47:40];
    DEST[31:24] := TEMP[39:32];
    DEST[39:32] := TEMP[31:24];
    DEST[47:40] := TEMP[23:16];
    DEST[55:48] := TEMP[15:8];
    DEST[63:56] := TEMP[7:0];

```

FI;

Flags Affected

None

Protected Mode Exceptions

#GP(0)	<p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#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	<p>If CPUID.01H:ECX.MOVBE[bit 22] = 0.</p> <p>If the LOCK prefix is used.</p> <p>If REP (F3H) prefix is used.</p>

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	<p>If CPUID.01H:ECX.MOVBE[bit 22] = 0.</p> <p>If the LOCK prefix is used.</p> <p>If REP (F3H) prefix is used.</p>

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 while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used. If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

MOVDIRI—Move Doubleword as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 F9 /r MOVDIRI m32, r32	A	V/V	MOVDIRI	Move doubleword from r32 to m32 using direct store.
NP REX.W + 0F 38 F9 /r MOVDIRI m64, r64	A	V/N.E.	MOVDIRI	Move quadword from r64 to m64 using direct store.

Instruction Operand Encoding¹

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	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 direct-store operation. The source operand is a general purpose register. The destination operand is a 32-bit memory location. 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 summary chart at the beginning of this section for encoding data and limits.

The direct-store is implemented by using write combining (WC) memory type protocol for writing data. 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. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store. Unlike stores with non-temporal hint that allow uncached (UC) and write-protected (WP) memory-type for the destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of the destination address memory type (including UC and WP types).

Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combining buffer may be combined with younger direct stores to the same address. **Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.**

Direct-stores issued by MOVDIRI to a destination aligned to a 4-byte boundary (8-byte boundary if used with REX.W prefix) guarantee 4-byte (8-byte with REX.W prefix) write-completion atomicity. This means that the data arrives at the destination in a single undivided 4-byte (or 8-byte) write transaction. If the destination is not aligned for the write size, the direct-stores issued by MOVDIRI are split and arrive at the destination in two parts. Each part of such split direct-store will not merge with younger stores but can arrive at the destination in either order. Availability of the MOVDIRI instruction is indicated by the presence of the CPUID feature flag MOVDIRI (bit 27 of the ECX register in leaf 07H, see "CPUID—CPU Identification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVDIRI void _directstoreu_u32(void *dst, uint32_t val)
MOVDIRI void _directstoreu_u64(void *dst, uint64_t val)
```

1. The Mod field of the ModR/M byte cannot have value 11B.

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.07H.0H:ECX.MOVDIRI[bit 27] = 0. If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0. If LOCK prefix or operand-size (66H) prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF (fault-code)	For a page fault.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0. If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

MOVDIR64B—Move 64 Bytes as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F8 /r MOVDIR64B r16/r32/r64, m512	A	V/V	MOVDIR64B	Move 64-bytes as direct-store with guaranteed 64-byte write atomicity from the source memory operand address to destination memory address specified as offset to ES segment in the register operand.

Instruction Operand Encoding¹

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

Description

Moves 64-bytes as direct-store with 64-byte write atomicity from source memory address to destination memory address. The source operand is a normal memory operand. The destination operand is a memory location specified in a general-purpose register. The register content is interpreted as an offset into ES segment without any segment override. In 64-bit mode, the register operand width is 64-bits (32-bits with 67H prefix). Outside of 64-bit mode, the register width is 32-bits when CS.D=1 (16-bits with 67H prefix), and 16-bits when CS.D=0 (32-bits with 67H prefix). MOVDIR64B requires the destination address to be 64-byte aligned. No alignment restriction is enforced for source operand.

MOVDIR64B first reads 64-bytes from the source memory address. It then performs a 64-byte direct-store operation to the destination address. The load operation follows normal read ordering based on source address memory-type. The direct-store is implemented by using the write combining (WC) memory type protocol for writing data. 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. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store.

Unlike stores with non-temporal hint which allow UC/WP memory-type for destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of destination address memory type (including UC/WP types). Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combining buffer may be combined with younger direct stores to the same address. **Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.**

There is no atomicity guarantee provided for the 64-byte load operation from source address, and processor implementations may use multiple load operations to read the 64-bytes. The 64-byte direct-store issued by MOVDIR64B guarantees 64-byte write-completion atomicity. This means that the data arrives at the destination in a single undivided 64-byte write transaction.

Availability of the MOVDIR64B instruction is indicated by the presence of the CPUID feature flag MOVDIR64B (bit 28 of the ECX register in leaf 07H, see "CPUID—CPU Identification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIR64B void _movdir64b(void *dst, const void* src)

1. The Mod field of the ModR/M byte cannot have value 11B.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If address in destination (register) operand is not aligned to a 64-byte boundary.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0. If 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. If address in destination (register) operand is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0. If 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 memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form. If address in destination (register) operand is not aligned to a 64-byte boundary.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0. If LOCK prefix is used.

MOVDQA, VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

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

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 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 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

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

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

Operation**VMOVDQA32 (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[MAXVL-1:VL] := 0

VMOVDQA32 (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;

VMOVDQA32 (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[MAXVL-1:VL] := 0

VMOVDQA64 (EVEX encoded versions, register-copy form)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := SRC[i+63:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE DEST[i+63:i] := 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VMOVDQA64 (EVEX encoded versions, store-form)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := SRC[i+63:i]

ELSE *DEST[i+63:i] remains unchanged* ; merging-masking

FI;

ENDFOR;

VMOVDQA64 (EVEX encoded versions, load-form)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := SRC[i+63:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE DEST[i+63:i] := 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VMOVDQA (VEX.256 encoded version, load - and register copy)

DEST[255:0] := SRC[255:0]

DEST[MAXVL-1:256] := 0

VMOVDQA (VEX.256 encoded version, store-form)

DEST[255:0] := SRC[255:0]

VMOVDQA (VEX.128 encoded version)

DEST[127:0] := SRC[127:0]

DEST[MAXVL-1:128] := 0

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

DEST[127:0] := SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

(V)MOVDQA (128-bit store-form version)

DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

```

VMOVDQA32 __m512i __mm512_load_epi32( void * sa);
VMOVDQA32 __m512i __mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i __mm512_maskz_load_epi32( __mmask16 k, void * sa);
VMOVDQA32 void __mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void __mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i __mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i __mm256_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void __mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i __mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i __mm_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void __mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i __mm512_load_epi64( void * sa);
VMOVDQA64 __m512i __mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i __mm512_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void __mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i __mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i __mm256_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void __mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i __mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i __mm_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void __mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i __mm256_load_si256 (__m256i * p);
MOVDQA __m256i __mm256_store_si256(__m256i *p, __m256i a);
MOVDQA __m128i __mm_load_si128 (__m128i * p);
MOVDQA void __mm_store_si128(__m128i *p, __m128i a);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C3 /r MOVNTI m32, r32	MR	V/V	SSE2	Move doubleword from r32 to m32 using non-temporal hint.
NP REX.W + OF C3 /r MOVNTI m64, r64	MR	V/N.E.	SSE2	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.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode / Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F D6 /r MOVQ2DQ <i>xmm, mm</i>	RM	V/V	SSE2	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

Instruction Operand Encoding

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

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM 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 MOVQ2DQ instruction is executed.

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

Operation

DEST[63:0] := SRC[63:0];

DEST[127:64] := 0000000000000000H;

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: `__128i_mm_movpi64_epi64 (__m64 a)`

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.
 #UD If CR0.EM[bit 2] = 1.
 If CR4.OSFXSR[bit 9] = 0.
 If CPUID.01H:EDX.SSE2[bit 26] = 0.
 If the LOCK prefix is used.
 #MF If there is a pending x87 FPU exception.

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.

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF BE /r	MOVSX r16, r/m8	RM	Valid	Valid	Move byte to word with sign-extension.
OF BE /r	MOVSX r32, r/m8	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + OF BE /r	MOVSX r64, r/m8	RM	Valid	N.E.	Move byte to quadword with sign-extension.
OF BF /r	MOVSX r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + OF BF /r	MOVSX r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r*	MOVSXD r16, r/m16	RM	Valid	N.E.	Move word to word with sign-extension.
63 /r*	MOVSXD r32, r/m32	RM	Valid	N.E.	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

* The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

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

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

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 := SignExtend(SRC);

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#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.
#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.

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 <i>mm1, mm2/m64</i>	A	V/V	MMX	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 0F 63 /r PACKSSWB <i>xmm1, xmm2/m128</i>	A	V/V	SSE2	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
NP 0F 6B /r ¹ PACKSSDW <i>mm1, mm2/m64</i>	A	V/V	MMX	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 0F 6B /r PACKSSDW <i>xmm1, xmm2/m128</i>	A	V/V	SSE2	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1, xmm2, xmm3/m128</i>	B	V/V	AVX	Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
VEX.128.66.0F.WIG 6B /r VPACKSSDW <i>xmm1, xmm2, xmm3/m128</i>	B	V/V	AVX	Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1, ymm2, ymm3/m256</i>	B	V/V	AVX2	Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.
VEX.256.66.0F.WIG 6B /r VPACKSSDW <i>ymm1, ymm2, ymm3/m256</i>	B	V/V	AVX2	Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.
EVEX.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into packed signed byte integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into packed signed byte integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.66.0F.WIG 63 /r VPACKSSWB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	C	V/V	AVX512BW	Converts packed signed word integers from <i>zmm2</i> and from <i>zmm3/m512</i> into packed signed byte integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.66.0F.WO 6B /r VPACKSSDW <i>xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</i>	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128/m32bcst</i> into packed signed word integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .

EVEX.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256/m32bcst</i> into packed signed word integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW	Converts packed signed doubleword integers from <i>zmm2</i> and from <i>zmm3/m512/m32bcst</i> into packed signed word integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .

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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	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.

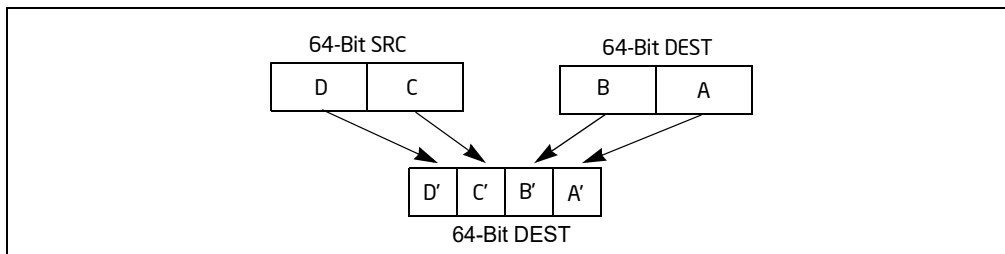


Figure 4-6. Operation of the PACKSSDW Instruction Using 64-bit Operands

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 word value is beyond the range of a signed byte value (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 (MAXVL-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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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[MAXVL-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 __mm128_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW __m128i __mm128_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 __mm128_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m128i __mm128_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.

POPF/POPFD/POPfq—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9D	POPF	Z0	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	Z0	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPfq	Z0	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHfq instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-15 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF¹, VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPfq (note that the 32-bit operand is not encodable). POPfq pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

Table 4-15. Effect of POPF/POPFQ on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																Notes	
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2		0
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF		CF
Real-Address Mode (CRO.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes (CRO.PE = 1 EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CRO.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CRO.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)	16	3	0-2	N/ X	N/ X	SV/ X	N/ X	N/ X	0/ X	S/ X	N/ X	S/ X	N/ X	S/ X	S/ X	S/ X	S/ X	S/ X	S/ X	S/ X	2,3
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

NOTES:

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

Key	
S	Updated from stack
SV	Updated from IF (bit 9) in FLAGS value on stack
N	No change in value
X	No EFLAGS update
0	Value is cleared

Operation

```

IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)
  THEN IF CPL = 0 OR CRO.PE = 0
    THEN
      IF OperandSize = 32;
        THEN
          EFLAGS := Pop(); (* 32-bit pop *)
          (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
             VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
        ELSE IF (OperandSize = 64)
          RFLAGS = Pop(); (* 64-bit pop *)
          (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
             VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
    
```

```

    ELSE (* OperandSize = 16 *)
        EFLAGS[15:0] := Pop(); (* 16-bit pop *)
        (* All non-reserved flags can be modified. *)
    FI;
ELSE (* CPL > 0 *)
    IF OperandSize = 32
        THEN
            IF CPL > IOPL
                THEN
                    EFLAGS := Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                    IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                ELSE
                    EFLAGS := Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                    IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
            FI;
        ELSE IF (OperandSize = 64)
            IF CPL > IOPL
                THEN
                    RFLAGS := Pop(); (* 64-bit pop *)
                    (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                    IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                ELSE
                    RFLAGS := Pop(); (* 64-bit pop *)
                    (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                    IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
            FI;
        ELSE (* OperandSize = 16 *)
            EFLAGS[15:0] := Pop(); (* 16-bit pop *)
            (* All non-reserved bits except IOPL can be modified; IOPL and all
            reserved bits are unaffected. *)
        FI;
    FI;
ELSE (* In virtual-8086 mode *)
    IF IOPL = 3
        THEN
            IF OperandSize = 32
                THEN
                    EFLAGS := Pop();
                    (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                    VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
                ELSE
                    EFLAGS[15:0] := Pop(); FI;
                    (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
            FI;
        ELSE (* IOPL < 3 *)
            IF (OperandSize = 32) OR (CR4.VME = 0)
                THEN #GP(0); (* Trap to virtual-8086 monitor. *)
            ELSE (* OperandSize = 16 and CR4.VME = 1 *)
                tempFLAGS := Pop();
                IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                    THEN #GP(0);
                ELSE

```

```

EFLAGS.VIF := tempFLAGS[9];
EFLAGS[15:0] := tempFLAGS;
(* All non-reserved bits except IOPL and IF can be modified;
IOPL, IF, and all reserved bits are unaffected. *)

```

FI;

FI;

FI;

FI;

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	<p>If IOPL < 3 and VME is not enabled.</p> <p>If IOPL < 3 and the 32-bit operand size is used.</p> <p>If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack.</p> <p>If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack.</p> <p>If an attempt is made to execute the POPF/POPFQ instruction with an operand-size override prefix.</p>
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

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	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2/m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2/m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3/m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	B	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.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	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 two unsigned word integer results.
EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	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.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	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 eight 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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	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 (MAXVL-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 (MAXVL-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 (MAXVL-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.

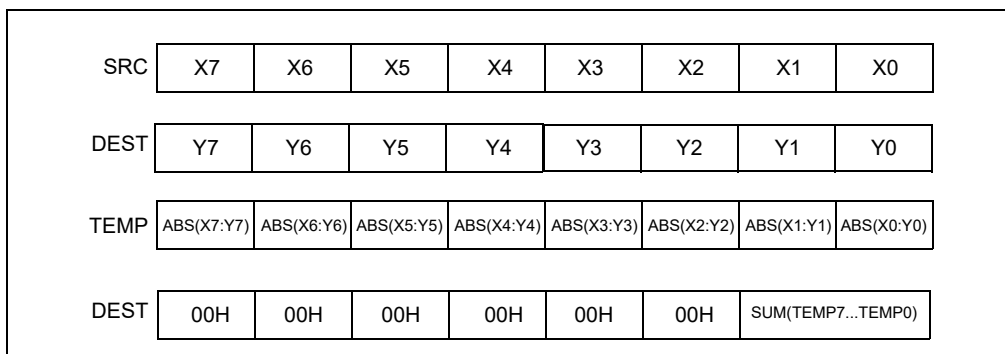


Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

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] := 000000000000H

```

```

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] := 000000000000H

```

```

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] := 000000000000H
  DEST[399:384] := SUM(TEMP16:TEMP23)
  DEST[447:400] := 000000000000H
  DEST[463:448] := SUM(TEMP24:TEMP31)
  DEST[511:464] := 000000000000H

```

```

FI;
DEST[MAXVL-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] := 000000000000H
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 000000000000H
DEST[MAXVL-1:256] := 0

```

VPSADBW (VEX.128 encoded version)

TEMPO := 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(TEMPO:TEMP7)
 DEST[63:16] := 000000000000H
 DEST[79:64] := SUM(TEMP8:TEMP15)
 DEST[127:80] := 000000000000H
 DEST[MAXVL-1:128] := 0

PSADBW (128-bit Legacy SSE version)

TEMPO := 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(TEMPO:TEMP7)
 DEST[63:16] := 000000000000H
 DEST[79:64] := SUM(TEMP8:TEMP15)
 DEST[127:80] := 000000000000
 DEST[MAXVL-1:128] (Unmodified)

PSADBW (64-bit operand)

TEMPO := 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(TEMPO: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.

PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9C	PUSHF	Z0	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	Z0	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	Z0	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFQ instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))

(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)

THEN

IF OperandSize = 32

THEN

push (EFLAGS AND 00FCFFFFH);

(* VM and RF bits are cleared in image stored on the stack *)

ELSE

push (EFLAGS); (* Lower 16 bits only *)

```

FI;
ELSE IF 64-bit MODE (* In 64-bit Mode *)
  IF OperandSize = 64
    THEN
      push (RFLAGS AND 00000000_00FCFFFFH);
      (* VM and RF bits are cleared in image stored on the stack; *)
    ELSE
      push (EFLAGS); (* Lower 16 bits only *)
  FI;
ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
  IF (CR4.VME = 0) OR (OperandSize = 32)
    THEN #GP(0); (* Trap to virtual-8086 monitor *)
  ELSE
    tempFLAGS = EFLAGS[15:0];
    tempFLAGS[9] = tempFLAGS[19]; (* VIF replaces IF *)
    tempFlags[13:12] = 3; (* IOPL is set to 3 in image stored on the stack *)
    push (tempFLAGS);
  FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

RET—Return from Procedure

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	Z0	Valid	Valid	Near return to calling procedure.
CB	RET	Z0	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	<i>imm16</i>	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.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for CET details.

Instruction ordering. Instructions following a far return may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far return have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Unlike near indirect CALL and near indirect JMP, the processor will not speculatively execute the next sequential instruction after a near RET unless that instruction is also the target of a jump or is a target in a branch predictor.

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();
        IF ShadowStackEnabled(CPL)
          tempSsEIP = ShadowStackPop4B();
          IF EIP != TempSsEIP
            THEN #CP(NEAR_RET); FI;
        FI;
      ELSE
        IF OperandSize = 64
          THEN
            IF top 8 bytes of stack not within stack limits
              THEN #SS(0); FI;
            RIP := Pop();
            IF ShadowStackEnabled(CPL)
              tempSsEIP = ShadowStackPop8B();
              IF RIP != tempSsEIP
                THEN #CP(NEAR_RET); FI;
            FI;
          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;
            IF ShadowStackEnabled(CPL)
              tempSsEIP = ShadowStackPop4B();
              IF EIP != tempSsEIP
                THEN #CP(NEAR_RET); FI;
            FI;
          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;

(* 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;
IF ShadowStackEnabled(CPL)
    (* SSP must be 8 byte aligned *)
    IF SSP AND 0x7 != 0
        THEN #CP(FAR-RET/IRET); FI;
tempSsCS = shadow_stack_load 8 bytes from SSP+16;
tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
prevSSP = shadow_stack_load 8 bytes from SSP;
SSP = SSP + 24;
(* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
tempCS = CS; (* zero pad to 64 bit *)
IF tempCS != tempSsCS
    THEN #CP(FAR-RET/IRET); FI;
(* do a 64 bit-compare; pad CSBASE+RIP with 0 for 32 bit LIP*)
IF CSBASE + RIP != tempSsLIP
    THEN #CP(FAR-RET/IRET); FI;
(* prevSSP must be 4 byte aligned *)

```

```

IF prevSSP AND 0x3 != 0
    THEN #CP(FAR-RET/IRET); FI;
(* In legacy mode SSP must be in low 4GB *)
IF prevSSP[63:32] != 0
    THEN #GP(0); FI;
SSP := prevSSP
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 *)
FI;
IF ShadowStackEnabled(CPL)
    (* check if 8 byte aligned *)
    IF SSP AND 0x7 != 0
        THEN #CP(FAR-RET/IRET); FI;
    IF ReturnCodeSegmentSelector(RPL) != 3
        THEN
            tempSsCS = shadow_stack_load 8 bytes from SSP+16;
            tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
            tempSSP = shadow_stack_load 8 bytes from SSP;
            SSP = SSP + 24;
            (* Do 64 bit compare to detect bits beyond 15 being set *)
            tempCS = CS; (* zero extended to 64 bit *)
            IF tempCS != tempSsCS
                THEN #CP(FAR-RET/IRET); FI;
            (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LA *)
            IF CSBASE + RIP != tempSsLIP
                THEN #CP(FAR-RET/IRET); FI;
            (* check if 4 byte aligned *)
            IF tempSSP AND 0x3 != 0
                THEN #CP(FAR-RET/IRET); FI;
        FI;
    FI;
FI;
tempOldCPL = CPL;

CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
    IF tempSSP[63:32] != 0
        THEN #GP(0); FI;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
FI;

FOR each SegReg in (ES, FS, GS, and DS)
    DO

```



```

tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
    THEN (* Segment register invalid *)
        SegmentSelector := 0; (*Segment selector becomes null*)

```

```

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;
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;
    FI;
IF ShadowStackEnabled(CPL)
    IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
        THEN #CP(FAR-RET/IRET); FI;
    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
    tempSSP = shadow_stack_load 8 bytes from SSP;
    SSP = SSP + 24;
    tempCS = CS; (* zero padded to 64 bit *)
    IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)

```

```

    THEN #CP(FAR-RET/IRET); FI;
IF CSBASE + RIP != tempSsLIP (* 64 bit compare *)
    THEN #CP(FAR-RET/IRET); FI;
IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
    THEN #CP(FAR-RET/IRET); FI;
IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
    (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP
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;
    IF ShadowStackEnabled(CPL)
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0

```

```

    THEN #CP(FAR-RET/IRET); FI;
IF ReturnCodeSegmentSelector(RPL) != 3
    THEN
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* Do 64 bit compare to detect bits beyond 15 being set *)
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
        IF CSBASE + RIP != tempSsLIP
            THEN #CP(FAR-RET/IRET); FI;
        (* check if 4 byte aligned *)
        IF tempSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
tempOldCPL = CPL;
CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
    IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
        (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
        THEN #GP(0); FI;
    SSP := tempSSP;
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT          (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                        (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
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;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If the return code or stack segment selector is NULL.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.</p>
#GP(selector)	<p>If the RPL of the return code segment selector is less than the CPL.</p> <p>If the return code or stack segment selector index is not within its descriptor table limits.</p> <p>If the return code segment descriptor does not indicate a code segment.</p> <p>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</p> <p>If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If the top bytes of stack are not within stack limits.</p> <p>If the return stack segment is not present.</p>
#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.
#CP(Far-RET/IRET)	<p>If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.</p> <p>If return instruction pointer from stack and shadow stack do not match.</p>

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)	<p>If the return instruction pointer is non-canonical.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return code segment selector is NULL.</p> <p>If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.</p>
#GP(selector)	<p>If the proposed segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p>
#NP(selector)	<p>If the return code or stack segment is not present.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>
#CP(Far-RET/IRET)	<p>If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.</p> <p>If return instruction pointer from stack and shadow stack do not match.</p>

RSTORSSP—Restore Saved Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 /5 (mod!=11, /5, memory only) RSTORSSP m64	M	V/V	CET_SS	Restore SSP.

Instruction Operand Encoding

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

Description

Restores SSP from the shadow-stack-restore token pointed to by m64. If the SSP restore was successful then the instruction replaces the shadow-stack-restore token with a previous-ssp token. The instruction sets the CF flag to indicate whether the SSP address recorded in the shadow-stack-restore token that was processed was 4 byte aligned, i.e., whether an alignment hole was created when the restore-shadow-stack token was pushed on this shadow stack.

Following RSTORSSP if a restore-shadow-stack token needs to be saved on the previous shadow stack, use the SAVEPREVSSP instruction.

If pushing a restore-shadow-stack token on the previous shadow stack is not required, the previous-ssp token can be popped using the INCSSPQ instruction. If the CF flag was set to indicate presence of an alignment hole, an additional INCSSPD instruction is needed to advance the SSP past the alignment hole.

Operation

IF CPL = 3

IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
THEN #UD; FI;

ELSE

IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
THEN #UD; FI;

FI;

SSP_LA = Linear_Address(mem operand)

IF SSP_LA not aligned to 8 bytes

THEN #GP(0); FI;

previous_ssp_token = SSP | (IA32_EFER.LMA AND CS.L) | 0x02

Start Atomic Execution

restore_ssp_token = Locked shadow_stack_load 8 bytes from SSP_LA

fault = 0

IF ((SSP_Tmp & 0x03) != (IA32_EFER.LMA & CS.L))

THEN fault = 1; FI; (* If L flag in token does not match IA32_EFER.LMA & CS.L or bit 1 is not 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND SSP_Tmp[63:32] != 0)

THEN fault = 1; FI; (* If compatibility/legacy mode and SSP to be restored not below 4G *)

TMP = restore_ssp_token & ~0x01

TMP = (TMP - 8)

TMP = TMP & ~0x07


```
IF TMP != SSP_LA
    THEN fault = 1; FI;    (* If address in token does not match the requested top of stack *)
```

```
TMP = (fault == 0) ? previous_ssp_token : restore_ssp_token
shadow_stack_store 8 bytes of TMP to SSP_LA and release lock
End Atomic Execution
```

```
IF fault == 1
    THEN #CP(RSTORSSP); FI;
```

```
SSP = SSP_LA
```

```
// Set the CF if the SSP in the restore token was 4 byte aligned, i.e., there is an alignment hole
RFLAGS.CF = (restore_ssp_token & 0x04) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;
```

Flags Affected

CF is set to indicate if the shadow stack pointer in the restore token was 4 byte aligned, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

C/C++ Compiler Intrinsic Equivalent

```
RSTORSSP void _rstorssp(void *);
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If destination is located in a non-writable segment. 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.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L). If address in token does not match linear address of memory operand. If in 32-bit or compatibility mode and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned. If a memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L). If address in token does not match linear address of memory operand.
#PF(fault-code)	If a page fault occurs.

SAVEPREVSSP—Save Previous Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EA (mod=11, /5, RM=010) SAVEPREVSSP	Z0	V/V	CET_SS	Save a restore-shadow-stack token on previous shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Push a restore-shadow-stack token on the previous shadow stack at the next 8 byte aligned boundary. The previous SSP is obtained from the previous-ssp token at the top of the current shadow stack.

Operation

IF CPL = 3

IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
THEN #UD; FI;

ELSE

IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
THEN #UD; FI;

FI;

IF SSP not aligned to 8 bytes

THEN #GP(0); FI;

(* Pop the "previous-ssp" token from current shadow stack *)

previous_ssp_token = ShadowStackPop8B(SSP)

(* If the CF flag indicates there was a alignment hole on current shadow stack then pop that alignment hole *)

(* Note that the alignment hole must be zero and can be present only when in legacy/compatibility mode *)

IF RFLAGS.CF == 1 AND (IA32_EFER.LMA AND CS.L)
#GP(0)

FI;

IF RFLAGS.CF == 1

must_be_zero = ShadowStackPop4B(SSP)
IF must_be_zero != 0 THEN #GP(0)

FI;

(* Previous SSP token must have the bit 1 set *)

IF ((previous_ssp_token & 0x02) == 0)
THEN #GP(0); (* bit 1 was 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND previous_ssp_token [63:32] != 0)
THEN #GP(0); FI; (* If compatibility/legacy mode and SSP not in 4G *)

(* Save Prev SSP from previous_ssp_token to the old shadow stack at next 8 byte aligned address *)

old_SSP = previous_ssp_token & ~0x03

temp := (old_SSP | (IA32_EFER.LMA & CS.L));

Shadow_stack_store 4 bytes of 0 to (old_SSP - 4)

old_SSP := old_SSP & ~0x07;

Shadow_stack_store 8 bytes of temp to (old_SSP - 8)

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

SAVEPREVSSP void _saveprevssp(void);

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If SSP not 8 byte aligned. If alignment hole on shadow stack is not 0. If bit 1 of the previous-ssp token is not set to 1. If in 32-bit/compatibility mode and SSP recorded in previous-ssp token is beyond 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. If CPL = 3 and IA32_U_CET.SH_STK_EN = 0. If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If SSP not 8 byte aligned. If carry flag is set. If bit 1 of the previous-ssp token is not set to 1.
#PF(fault-code)	If a page fault occurs.

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	A	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	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 (MAXVL-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 (MAXVL-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[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

VSQRTSS (VEX.128 encoded version)

```

DEST[31:0] := SQRT(SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

SQRTSS (128-bit Legacy SSE version)

```

DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-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.

SYSCALL—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 05	SYSCALL	Z0	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

When shadow stacks are enabled at a privilege level where the SYSCALL instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional CET details.

Instruction ordering. Instructions following a SYSCALL may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSCALL have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
 (* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)

THEN #UD;

FI;

RCX := RIP; (* Will contain address of next instruction *)

RIP := IA32_LSTAR;

R11 := RFLAGS;

RFLAGS := RFLAGS AND NOT(IA32_FMASK);

CS.Selector := IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)

(* Set rest of CS to a fixed value *)

CS.Base := 0; (* Flat segment *)

```

CS.Limit := FFFFFFFH;          (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;                 (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 0;
CS.P := 1;
CS.L := 1;                     (* Entry is to 64-bit mode *)
CS.D := 0;                     (* Required if CS.L = 1 *)
CS.G := 1;                     (* 4-KByte granularity *)

```

```
IF ShadowStackEnabled(CPL)
```

```
  THEN (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
```

```
    IA32_PL3_SSP := LA_adjust(SSP);
```

```
      (* With shadow stacks enabled the system call is supported from Ring 3 to Ring 0 *)
```

```
      (* OS supporting Ring 0 to Ring 0 system calls or Ring 1/2 to ring 0 system call *)
```

```
      (* Must preserve the contents of IA32_PL3_SSP to avoid losing ring 3 state *)
```

```
FI;
```

```
CPL := 0;
```

```
IF ShadowStackEnabled(CPL)
```

```
  SSP := 0;
```

```
FI;
```

```
IF EndbranchEnabled(CPL)
```

```
  IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
```

```
  IA32_S_CET.SUPPRESS = 0
```

```
FI;
```

```
SS.Selector := IA32_STAR[47:32] + 8;      (* SS just above CS *)
```

```
(* Set rest of SS to a fixed value *)
```

```
SS.Base := 0;                             (* Flat segment *)
```

```
SS.Limit := FFFFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
```

```
SS.Type := 3;                             (* Read/write data, accessed *)
```

```
SS.S := 1;
```

```
SS.DPL := 0;
```

```
SS.P := 1;
```

```
SS.B := 1;                                (* 32-bit stack segment *)
```

```
SS.G := 1;                                (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD If IA32_EFER.SCE = 0.
 If the LOCK prefix is used.

SYSENTER—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 34	SYSENTER	Z0	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; FI;
    ELSE
      SYSENTER/SYSEXIT_Supported; FI;
  FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level where SYSENTER instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional CET details.

Instruction ordering. Instructions following a SYSENTER may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSENTER have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM := 0;                (* Ensures protected mode execution *)
RFLAGS.IF := 0;                (* Mask interrupts *)
IF in IA-32e mode
  THEN
    RSP := IA32_SYSENTER_ESP;
    RIP := IA32_SYSENTER_EIP;
  ELSE
    ESP := IA32_SYSENTER_ESP[31:0];
    EIP := IA32_SYSENTER_EIP[31:0];
  FI;

CS.Selector := IA32_SYSENTER_CS[15:0] AND FFFCH;
                                     (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;                    (* Flat segment *)
CS.Limit := FFFFFFFH;           (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;                  (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 0;
CS.P := 1;
IF in IA-32e mode
  THEN
    CS.L := 1;                   (* Entry is to 64-bit mode *)
    CS.D := 0;                   (* Required if CS.L = 1 *)
  ELSE
    CS.L := 0;
    CS.D := 1;                   (* 32-bit code segment*)
```

```

FI;
CS.G := 1;                                (* 4-KByte granularity *)

IF ShadowStackEnabled(CPL)
  THEN
    IF IA32_EFER.LMA = 0
      THEN IA32_PL3_SSP := SSP;
      ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
        IA32_PL3_SSP := LA_adjust(SSP);
    FI;
  FI;

CPL := 0;

IF ShadowStackEnabled(CPL)
  SSP := 0;
FI;
IF EndbranchEnabled(CPL)
  IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
  IA32_S_CET.SUPPRESS = 0
FI;

SS.Selector := CS.Selector + 8;            (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base := 0;                             (* Flat segment *)
SS.Limit := FFFFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3;                             (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 0;
SS.P := 1;
SS.B := 1;                                (* 32-bit stack segment*)
SS.G := 1;                                (* 4-KByte granularity *)

```

Flags Affected

VM, IF (see Operation above)

Protected Mode Exceptions

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

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

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 35	SYSEXIT	Z0	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	Z0	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protection levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; FI;
    ELSE
      SYSENTER/SYSEXIT_Supported; FI;
  FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR. Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 18, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for additional CET details.

Instruction ordering. Instructions following a SYSEXIT may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSEXIT have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF IA32_SYSENTER_CS[15:2] = 0 OR CRO.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;
```

```
IF operand size is 64-bit
```

```
    THEN (* Return to 64-bit mode *)
```

```
        RSP := RCX;
```

```
        RIP := RDX;
```

```
    ELSE (* Return to protected mode or compatibility mode *)
```

```
        RSP := ECX;
```

```
        RIP := EDX;
```

```
FI;
```

```
IF operand size is 64-bit (* Operating system provides CS; RPL forced to 3 *)
```

```
    THEN CS.Selector := IA32_SYSENTER_CS[15:0] + 32;
```

```
    ELSE CS.Selector := IA32_SYSENTER_CS[15:0] + 16;
```

```
FI;
```

```
CS.Selector := CS.Selector OR 3; (* RPL forced to 3 *)
```

```
(* Set rest of CS to a fixed value *)
```

```
CS.Base := 0; (* Flat segment *)
```

```
CS.Limit := FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
```

```
CS.Type := 11; (* Execute/read code, accessed *)
```

```
CS.S := 1;
```

```
CS.DPL := 3;
```

```
CS.P := 1;
```

```
IF operand size is 64-bit
```

```
    THEN (* return to 64-bit mode *)
```

```
        CS.L := 1; (* 64-bit code segment *)
```

```
        CS.D := 0; (* Required if CS.L = 1 *)
```

```
    ELSE (* return to protected mode or compatibility mode *)
```

```
        CS.L := 0;
```

```
        CS.D := 1; (* 32-bit code segment *)
```

```
FI;
```

```
CS.G := 1; (* 4-KByte granularity *)
```

```
CPL := 3;
```

```
IF ShadowStackEnabled(CPL)
```

```
    THEN SSP := IA32_PL3_SSP;
```

```
FI;
```

```
SS.Selector := CS.Selector + 8; (* SS just above CS *)
```

```
(* Set rest of SS to a fixed value *)
```

```
SS.Base := 0; (* Flat segment *)
```

```
SS.Limit := FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
```

```
SS.Type := 3; (* Read/write data, accessed *)
```

```
SS.S := 1;
```

```
SS.DPL := 3;
```

SS.P := 1;
 SS.B := 1; (* 32-bit stack segment*)
 SS.G := 1; (* 4-KByte granularity *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
 If CPL ≠ 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP The SYSEXIT instruction is not recognized in real-address mode.
 #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If IA32_SYSENTER_CS = 0.
 If CPL ≠ 0.
 If RCX or RDX contains a non-canonical address.
 #UD If the LOCK prefix is used.

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	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 operation 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 OF 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	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	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	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 operation 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.

9. Updates to Chapter 5, Volume 2C

Change bars and green text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z*.

Changes to this chapter include updates to various instructions. Updates to the following instructions: VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTPD, VGETMANTPS, VGETMANTSD, VGETMANTSS, VPDPWSSD, VPDPWSSDS, VPMADD52HUQ, VPMADD52LUQ, and WRPKRU.

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where $x = \text{approx}(1/0)$, yields an incorrect result. To deal with this, `VFIXUPIMMSS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}
```

```

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
  tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j := 0;
    SNAN_TOKEN: j := 1;
    ZERO_VALUE_TOKEN: j := 2;
    POS_ONE_VALUE_TOKEN: j := 3;
    NEG_INF_TOKEN: j := 4;
    POS_INF_TOKEN: j := 5;
    NEG_VALUE_TOKEN: j := 6;
    POS_VALUE_TOKEN: j := 7;
  } ; end source special CASE(tsrc...)

```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```

CASE(token_response[3:0]) {
  0000: dest[31:0] := dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] := tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] := QNaN(tsrc[31:0]);
  0011: dest[31:0] := QNaN_Indefinite;
  0100: dest[31:0] := -INF;
  0101: dest[31:0] := +INF;
  0110: dest[31:0] := tsrc.sign? -INF : +INF;
  0111: dest[31:0] := -0;
  1000: dest[31:0] := +0;
  1001: dest[31:0] := -1;
  1010: dest[31:0] := +1;
  1011: dest[31:0] := 1/2;
  1100: dest[31:0] := 90.0;
  1101: dest[31:0] := PI/2;
  1110: dest[31:0] := MAX_FLOAT;
  1111: dest[31:0] := -MAX_FLOAT;
} ; end of token_response CASE

```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
```

```
 ; end fault reporting
```

```
return dest[31:0];
```

```
} ; end of FIXUPIMM_SP()
```

VFIXUPIMMSS (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] := FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE DEST[31:0] := 0 ; zeroing-masking
    FI
  FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

Immediate Control Description:

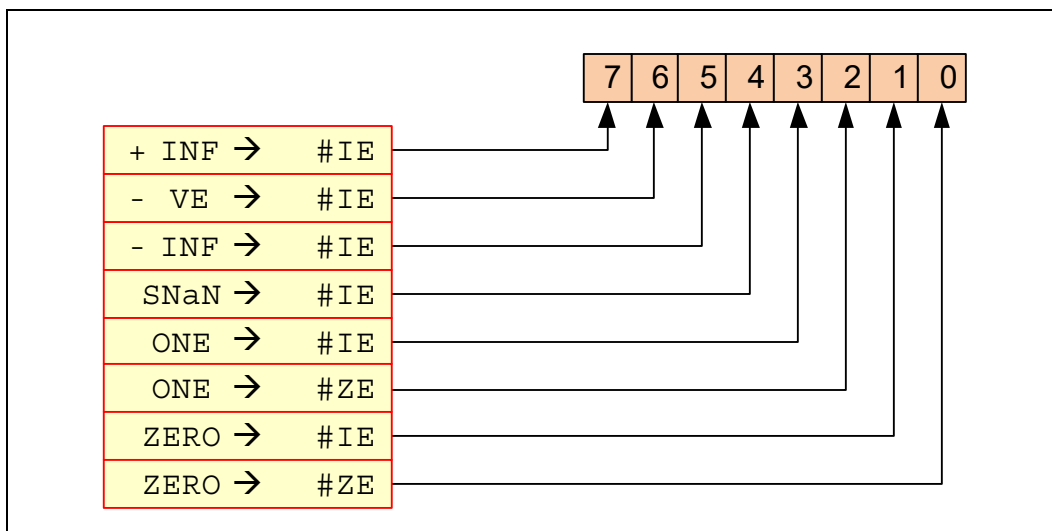


Figure 5-12. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

```

VFIXUPIMMSS __m128 __mm_fixupimm_ss(__m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_fixupimm_round_ss(__m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_round_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

```

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.

VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-5.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

VGETEXPSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
        ConvertExpDPFP(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[MAXVL-1:128] := 0
```


Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);  
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);  
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS

VGETEXPSS (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] :=
    ConvertExpDPFP(SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[31:0] := 0
    FI
  FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);  
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);  
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);  
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);  
VGETEXPSS __m128 _mm_mask_getexp_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int sae);  
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask *k1*. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. 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 64-bit memory location.

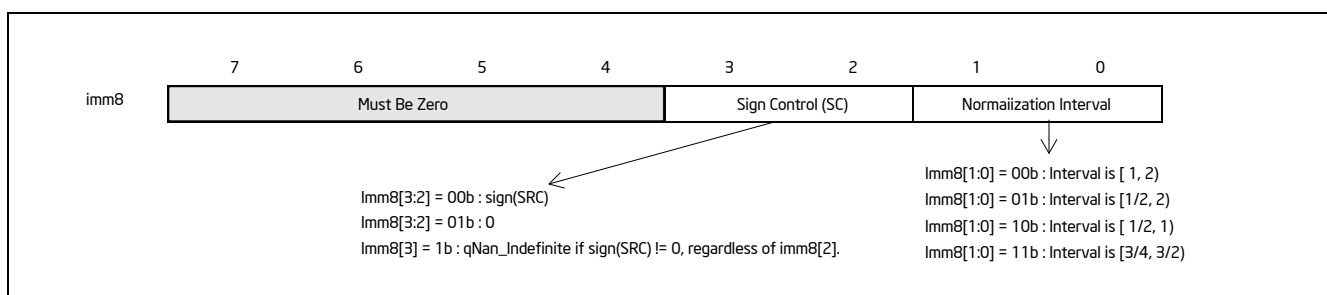


Figure 5-15. *Imm8* Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value *x*, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If $interv \neq 0$ then $k = -1$, otherwise $k = 0$. The encoded value of $imm8[1:0]$ and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by $interv$.

The `GetMant()` function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register $k1$ are computed and stored into the destination. Elements in $zmm1$ with the corresponding bit clear in $k1$ retain their previous values.

Note: `EVEX.vvvv` is reserved and must be `1111b`; otherwise instructions will `#UD`.

Table 5-7. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore <i>interv</i> If (SRC = SNaN) then #IE
$+\infty$	1.0	Ignore <i>interv</i>
+0	1.0	Ignore <i>interv</i>
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i>
$-\infty$	IF (SC[1]) THEN {QNaN_Indefinite} ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i> If (SC[1]) then #IE
negative	SC[1] ? QNaN_Indefinite : Getmant(SRC) ¹	If (SC[1]) then #IE

NOTES:

1. In case $SC[1] \neq 0$, the sign of `Getmant(SRC)` is declared according to $SC[0]$.

Operation

```
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
```

```

        return signed_one
    if infinity:
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite
        return signed_one
    if sign_control[1]:
        MXCSR.IE := 1
        return QNaN_Indefinite

if denormal:
    if MXCSR.DAZ:
        dst.fraction := 0
    else:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[51]
            dst.fraction := dst.fraction << 1
            dst.exp := dst.exp - 1
        MXCSR.DE := 1

unbiased_exp := dst.exp - bias
odd_exp := unbiased_exp[0]
signaling_bit := dst.fraction[51]
if normalization_interval = 0b00:
    dst.exp := bias
else if normalization_interval = 0b01:
    dst.exp := odd_exp ? bias-1 : bias
else if normalization_interval = 0b10:
    dst.exp := bias-1
else if normalization_interval = 0b11:
    dst.exp := signaling_bit ? bias-1 : bias
return dst

```

VGETMANTPD (EVEX encoded versions)

VGETMANTPD dest{k1}, src, imm8

VL = 128, 256, or 512

KL := VL / 64

sign_control := imm8[3:2]

normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:

IF k1[i] or *no writemask*:

IF SRC is memory and (EVEX.b = 1):

tsrc := src.double[0]

ELSE:

tsrc := src.double[i]

DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)

ELSE IF *zeroing*:

DEST.double[i] := 0

//else DEST.double[i] remains unchanged

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. 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.

For each input SP FP value x , The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then $k = -1$, otherwise $k = 0$. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```

def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (dst.fraction = 0)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
    nan := (dst.exp = 0xFF) and (dst.fraction != 0)
    src_signaling := src.fraction[22]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        if MXCSR.DAZ:
            dst.fraction := 0
        else:
            jbit := 0
            dst.exp := bias
            while jbit = 0:
                jbit := dst.fraction[22]
                dst.fraction := dst.fraction << 1
                dst.exp := dst.exp - 1
            MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[22]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:

```

```

dst.exp := bias-1
else if normalization_interval = 0b11:
    dst.exp := signaling_bit ? bias-1 : bias
return dst

```

VGETMANTPS (EVEX encoded versions)

VGETMANTPS dest{k1}, src, imm8

VL = 128, 256, or 512

KL := VL / 32

sign_control := imm8[3:2]

normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:

IF k1[i] or *no writemask*:

IF SRC is memory and (EVEX.b = 1):

tsrc := src.float[0]

ELSE:

tsrc := src.float[i]

DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)

ELSE IF *zeroing*:

DEST.float[i] := 0

//else DEST.float[i] remains unchanged

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPS __m512 __mm512_getmant_ps(__m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_getmant_round_ps(__m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m512 __mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m512 __mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m256 __mm256_getmant_ps(__m256 a, enum intv, enum sgn);

VGETMANTPS __m256 __mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);

VGETMANTPS __m256 __mm256_maskz_getmant_ps(__mmask8 k, __m256 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm_getmant_ps(__m128 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm_maskz_getmant_ps(__mmask8 k, __m128 a, enum intv, enum sgn);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa of the low float64 element in xmm3/m64 using <i>imm8</i> for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If *interv* != 0 then *k* = -1, otherwise *k* = 0. The encoded value of *imm8*[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent *k* (adding bias) and a mantissa normalized to the range specified by *interv*.

The *GetMant()* function follows Table 5-7 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Operation

`// getmant_fp64(src, sign_control, normalization_interval)` is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)

SignCtrl[1:0] := IMM8[3:2];

Interv[1:0] := IMM8[1:0];

IF k1[0] OR *no writemask*

THEN DEST[63:0] :=

`getmant_fp64(src, sign_control, normalization_interval)`

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[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSD `__m128d __mm_getmant_sd(__m128d a, __m128 b, enum intv, enum sgn);`

VGETMANTSD `__m128d __mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);`

VGETMANTSD `__m128d __mm_maskz_getmant_sd(__mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);`

VGETMANTSD `__m128d __mm_getmant_round_sd(__m128d a, __m128 b, enum intv, enum sgn, int r);`

VGETMANTSD `__m128d __mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);`

VGETMANTSD `__m128d __mm_maskz_getmant_round_sd(__mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);`

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv != 0 then $k = -1$, otherwise $k = 0$. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Operation

// `getmant_fp32(src, sign_control, normalization_interval)` is defined in the operation section of `VGETMANTPS`

VGETMANTSS (EVEX encoded version)

```

SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
  THEN DEST[31:0] :=
    getmant_fp32(src, sign_control, normalization_interval)
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[31:0] := 0
    FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTSS __m128 __mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 __mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 __mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VPDPWSSD – Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2

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

ORIGDEST := DEST

FOR i := 0 TO KL-1:

IF k1[i] or *no writemask*:

IF SRC2 is memory and EVEX.b == 1:

t := SRC2.dword[0]

ELSE:

t := SRC2.dword[i]

p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])

p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])

DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword

ELSE IF *zeroing*:

DEST.dword[i] := 0

ELSE: // Merge masking, dest element unchanged

DEST.dword[i] := ORIGDEST.dword[i]

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);  
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);  
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);  
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);  
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);  
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);  
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);  
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);  
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPDPWSSDS – Multiply and Add Signed Word Integers with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers or 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2

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

ORIGDEST := DEST

FOR i := 0 TO KL-1:

IF k1[i] or *no writemask*:

IF SRC2 is memory and EVEX.b == 1:

t := SRC2.dword[0]

ELSE:

t := SRC2.dword[i]

p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])

p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])

DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)

ELSE IF *zeroing*:

DEST.dword[i] := 0

ELSE: // Merge masking, dest element unchanged

DEST.dword[i] := ORIGDEST.dword[i]

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

```

VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators

Opcode/ Instruction	Op/ En	32/64 bit Mode Support	CPUID	Description
EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	NA

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation**VPMADD52HUQ (EVEX encoded)**

(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 src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] := ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] := ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];

Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]);

DEST[i+63:i] := Temp2[63:0];

ELSE

IF *zeroing-masking* THEN

DEST[i+63:i] := 0;

ELSE *merge-masking*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52HUQ __m512i __mm512_madd52hi_epu64(__m512i a, __m512i b, __m512i c);

VPMADD52HUQ __m512i __mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);

VPMADD52HUQ __m512i __mm512_maskz_madd52hi_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);

VPMADD52HUQ __m256i __mm256_madd52hi_epu64(__m256i a, __m256i b, __m256i c);

VPMADD52HUQ __m256i __mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);

VPMADD52HUQ __m256i __mm256_maskz_madd52hi_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);

VPMADD52HUQ __m128i __mm_madd52hi_epu64(__m128i a, __m128i b, __m128i c);

VPMADD52HUQ __m128i __mm_mask_madd52hi_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);

VPMADD52HUQ __m128i __mm_maskz_madd52hi_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators

Opcode/ Instruction	Op/En	32/64 bit Mode Support	CPUID	Description
EVEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	NA

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation**VPMADD52LUQ (EVEX encoded)**

(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 src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] := ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] := ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];

Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[51:0]);

DEST[i+63:i] := Temp2[63:0];

ELSE

IF *zeroing-masking* THEN

DEST[i+63:i] := 0;

ELSE *merge-masking*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX_VL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52LUQ __m512i __mm512_madd52lo_epu64(__m512i a, __m512i b, __m512i c);

VPMADD52LUQ __m512i __mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);

VPMADD52LUQ __m512i __mm512_maskz_madd52lo_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);

VPMADD52LUQ __m256i __mm256_madd52lo_epu64(__m256i a, __m256i b, __m256i c);

VPMADD52LUQ __m256i __mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);

VPMADD52LUQ __m256i __mm256_maskz_madd52lo_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);

VPMADD52LUQ __m128i __mm_madd52lo_epu64(__m128i a, __m128i b, __m128i c);

VPMADD52LUQ __m128i __mm_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);

VPMADD52LUQ __m128i __mm_maskz_madd52lo_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

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	Z0	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	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.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKRU have completed execution and updated the PKRU register.

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.

10. Updates to Chapter 6, Volume 2D

Change bars and green text show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference*.

Changes to this chapter include updates to various instructions. Updates to the following instructions: GETSEC[EXITAC] and GETSEC[SENDER].

GETSEC[EXITAC]—Exit Authenticated Code Execution Mode

Opcode	Instruction	Description
NP OF 37 (EAX=3)	GETSEC[EXITAC]	Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags.

Description

The GETSEC[EXITAC] leaf function exits the ILP out of authenticated code execution mode established by GETSEC[ENTERACCS] or GETSEC[SENDER]. The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC[EXITAC] can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC[EXITAC] operation, the processor unmask responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC[SENDER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENDER], then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC[SEXIT]. INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC[EXITAC] instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. It should be cautioned that if no VMX environment is to be activated following GETSEC[EXITAC], that these events will remain masked until the measured environment is exited with GETSEC[SEXIT]. If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC[EXITAC]. Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.

Upon completion of GETSEC[EXITAC] a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

Operation

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF ( in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or ( in 64-bit mode) and ( RBX is non-canonical) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
    THEN #GP(0);
IF (OperandSize = 32)
    THEN tempEIP := EBX;
ELSIF (OperandSize = 64)
    THEN tempEIP := RBX;
ELSE
    tempEIP := EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
    THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENERFLAG=0)
    THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ACMODEFLAG := 0;
IF IA32_EFER.LMA == 1
    THEN CR3 := R8;
EIP := tempEIP;
END;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX.W	Sets 64-bit mode Operand size attribute.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1. If in VMX root operation. If the processor is not currently in authenticated code execution mode. If the processor is in SMM. If any reserved bit position is set in the EDX parameter register.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[EXITAC] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[EXITAC] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP(0)	If the target address in RBX is not in a canonical form.
--------	--

VM-Exit Condition

Reason (GETSEC)	IF in VMX non-root operation.
-----------------	-------------------------------

GETSEC[SENTER]—Enter a Measured Environment

Opcode	Instruction	Description
NP OF 37 (EAX=4)	GETSEC[SENTER]	Launch a measured environment. EBX holds the SINIT authenticated code module physical base address. ECX holds the SINIT authenticated code module size (bytes). EDX controls the level of functionality supported by the measured environment launch.

Description

The GETSEC[SENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENTER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[SENTER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS_0.activeLocality bit is clear before executing the GETSEC[SENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENTER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAMETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[SENTER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.

A successful launch of the measured environment results in the initiating logical processor entering the authenticated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.
- Establish and check the location and size of the authenticated code module to be executed by the ILP.
- Check for the existence of an Intel® TXT-capable chipset.
- Verify the current power management configuration is acceptable.
- Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
- Load the designated AC module into authenticated code execution area.
- Isolate the content of authenticated code execution area from further state modification by external agents.
- Authenticate the AC module.
- Updated the Trusted Platform Module (TPM) with the authenticated code module's hash.
- Initialize processor state based on the authenticated code module header information.
- Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENDER] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCI_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendezvous message. Any outstanding valid uncorrectable machine check error condition present in the machine check status registers at the first check point will result in the ILP signaling a general protection violation. If an outstanding valid uncorrectable machine check error condition is present at the second check point, then this will result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR IA32_PERF_STATUS values are compared against either the processor supported maximum operating target setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER. The measured environment software may need to take responsibility for restoring such settings that are deemed to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of the external software to program the chipset voltage ID and/or bus ratio select settings to known good values recognized by the processor, prior to executing SENTER.

NOTE

For a mobile processor, an adjustment can be made according to the thermal monitor operating target. For a quad-core processor the SENTER adjustment mechanism may result in a more conservative but non-uniform voltage setting, depending on the pre-SENDER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally or unconditionally via the GETSEC[EXITAC], GETSEC[SEXIT], GETSEC[SMCTRL] or for specific VMX related operations such as a VM entry or the VMXOFF instruction (see respective GETSEC leaves and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* for more details). The state of the A20M pin is masked and forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by

GETSEC[SENDER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERRORCODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in a trusted storage facility in the platform. The following trusted storage facilities are supported:

- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 0, the computed hash is stored to the platform's TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENDER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules.
- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM's PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to EBX + [GDTBasePtr] and GDTR limit set to [GDTLimit]. The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFh, G=1, D=1, P=1, S=1, read/write/accessed for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to EBX + [EntryPoint]. AC module defined fields used for initializing processor state are consistency checked with a failure resulting in an TXT-shutdown condition.

Table 6-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENDER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.

Table 6-6. Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CR0	PG←0, AM←0, WP←0; Others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header ¹]	[LT.MLE.JOIN + 12]
EBX	Unchanged [SINIT.BASE]	Unchanged
EDX	SENTER control flags	Unchanged
EBP	SINIT.BASE	Unchanged
CS	Sel=[SINIT SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFH, G = 1, D = 1, AR = 9BH
DS, ES, SS	Sel=[SINIT SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	Sel = [LT.MLE.JOIN + 8] +8, base = 0, limit = FFFFFH, G = 1, D = 1, AR = 93H
GDTR	Base= SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL	0H	0H
Performance counters and counter control registers	0H	0H
IA32_MISC_ENABLE	See Table 6-5	See Table 6-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

NOTES:

1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for MLE header format.

Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGS, and the MSR IA32_DEBUGCTL as defined in Table 6-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-

shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32_MISC_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 6-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

Effect of MSR IA32_FEATURE_CONTROL MSR

Bits 15:8 of the IA32_FEATURE_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32_FEATURE_CONTROL MSR is shown in Table 6-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32_FEATURE_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32_FEATURE_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32_FEATURE_CONTROL is treated as reserved.

The *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SENTER] (ILP only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
(CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
(CPL>0) or (EFLAGS.VM=1) or
(IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
(SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
(TPM interface is not present) or
(EDX ≠ (SENTER_EDX_support_mask & EDX)) or
(IA32_FEATURE_CONTROL[0]=0) or (IA32_FEATURE_CONTROL[15]=0) or
((IA32_FEATURE_CONTROL[14:8] & EDX[6:0]) ≠ EDX[6:0]))
    THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF IA32_MC[I]_STATUS = uncorrectable error
            THEN #GP(0);
    FI;
OD;
```

```

FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN #GP(0);
ACBASE := EBX;
ACSIZE := ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0) or (ACSIZE < minimum
  module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
  THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
WHILE (no SignalSENTER message);

```

TXT_SENTER__MSG_EVENT (ILP & RLP):

```

Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
  THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
  IF IA32_MC[I]_STATUS = uncorrectable error
    THEN TXT-SHUTDOWN(#UnrecovMCErr);
  FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN TXT-SHUTDOWN(#UnrecovMCErr);
IF (Voltage or bus ratio status are NOT at a known good state)
  THEN IF (Voltage select and bus ratio are internally adjustable)
    THEN
      Make product-specific adjustment on operating parameters;
    ELSE
      TXT-SHUTDOWN(#IllegalVIDBRatio);
  FI;

```

```

IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M := 0;
IA32_DEBUGCTL := 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG := 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
  THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
  DONE := TXT.READ(LT.STS);
WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACBASE+ACSIZE-1 DO
  ACRAM[I-ACBASE].ADDR := I;
  ACRAM[I-ACBASE].DATA := LOAD(I);
OD;

```

```

IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified
    line detected on ACRAM load))
    THEN ACEntryPoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint := ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);

IF (FTM_INTERFACE_ID.[3:0] = 1 ) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at 0FED44008H, TMP_DATA_BUFFER_4 is located at 0FED44080H *)
        WRITE(TPM_LOC_CTRL_4) := 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4) := SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I) := ACRAM[SCRATCH.I];
            WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST) := EDX;
            WRITE(FTM.LOC_CTRL) := 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
        ELSE IF (FTM_INTERFACE_ID.[3:0] = 0 ) (* Use standard TPM Interface *)
            ACRAM[SCRATCH.SIGNATURE_LEN_CONST] := EDX;
            WRITE(TPM.HASH.START) := 0;
            FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
                WRITE(TPM.HASH.DATA) := ACRAM[SCRATCH.I];
                WRITE(TPM.HASH.END) := 0;
FI;

```

```

ACMODEFLAG := 1;
CR0.[PG.AM.WP] := 0;
CR4 := 00004000h;
EFLAGS := 00000002h;
IA32_EFER := 0;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
DS.BASE := 0;
DS.LIMIT := FFFFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
SS := DS;
ES := DS;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;

```

RLP_SENTER_ROUTINE: (RLP only)

```

Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP := 0;
GOTO SENTER sleep state;
END;

```

Flags Affected

All flags are cleared.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1. If in VMX root operation. If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP. If an Intel® TXT-capable chipset is not present. If an Intel® TXT-capable chipset interface to TPM is not detected as present. If a protected partition is already active or the processor is already in authenticated code mode. If the processor is in SMM. If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS. If the authenticated code base is not on a 4096 byte boundary. If the authenticated code size > processor's authenticated code execution area storage capacity. If the authenticated code size is not modulo 64.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SENTER] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SENTER] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP	IF AC code module does not reside in physical address below $2^{32} - 1$.
-----	--

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP	IF AC code module does not reside in physical address below $2^{32} - 1$.
-----	--

VM-Exit Condition

Reason (GETSEC)	IF in VMX non-root operation.
-----------------	-------------------------------

11. Updates to Chapter 7, Volume 2D

Change bars and green text show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference*.

Changes to this chapter include updates to the V4FMADDPS/V4FNMADDPS instructions.

V4FMADDPS/V4FNMADDPS — Packed Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 9A /r V4FMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.
EVEX.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
```

```
    tmpdest := dest
```

```
    // reg[] is an array representing the SIMD register file.
```

```
    FOR j := 0 to regs_loaded-1:
```

```
        FOR i := 0 to kl-1:
```

```
            IF k1[i] or *no writemask*:
```

```
                IF posneg = 0:
```

```
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] - reg[src_base + j].single[i] * msrc.single[j])
```

```
                ELSE:
```

```
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j].single[i] * msrc.single[j])
```

```
            ELSE IF *zeroing*:
```

```
                tmpdest.single[i] := 0
```

```
    dest := tmpdst
```

```
    dest[MAX_VL-1:VL] := 0
```

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)

KL, VL = (16,512)

regs_loaded := 4

src_base := src_reg_id & ~3 // for src1 operand

posneg := 0 if negative form, 1 otherwise

NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)

Intel C/C++ Compiler Intrinsic Equivalent

V4FMADDPS __m512 _mm512_4fmadd_ps(__m512, __m512x4, __m128 *);

V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);

V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);

V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);

V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);

V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally

#UD If the EVEX broadcast bit is set to 1.

#UD If the MODRM.mod = 0b11.

12. Updates to Appendix A, Volume 2D

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

Changes to this chapter include typo corrections.

Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined. Cells marked “Reserved-NOP” are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked “Reserved-NOP” nor depend on the current behavior of those opcodes.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- B The VEX.vvvv field of the VEX prefix selects a general purpose register.

- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).
- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS/RFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.
- I Immediate data: the operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.
- O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.

dq	Double-quadword, regardless of operand-size attribute.
p	32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd	128-bit or 256-bit packed double-precision floating-point data.
pi	Quadword MMX technology register (for example: mm0).
ps	128-bit or 256-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
qq	Quad-Quadword (256-bits), regardless of operand-size attribute.
s	6-byte or 10-byte pseudo-descriptor.
sd	Scalar element of a 128-bit double-precision floating data.
ss	Scalar element of a 128-bit single-precision floating data.
si	Doubleword integer register (for example: eax).
v	Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w	Word, regardless of operand-size attribute.
x	dq or qq based on the operand-size attribute.
y	Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.

- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSH A; 06H, PUSH ES).

Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA405000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
 - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
 - Gv: The reg field of the ModR/M byte selects a general-purpose register.
 - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).

- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:0000000H, EAX, 3.

A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3AH. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
 - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
 - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
 - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix "v" is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table A-1. Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction), the 0FB9H opcode (UD1 instruction), or the 0FFFH opcode (UDO instruction) when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH ES ⁶⁴	POP ES ⁶⁴
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH SS ⁶⁴	POP SS ⁶⁴
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=ES (Prefix)	DAA ⁶⁴
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=SS (Prefix)	AAA ⁶⁴
4	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH ⁶⁴ _A / PUSH ⁶⁴ _D	POP ⁶⁴ _A / POP ⁶⁴ _D	BOUND ⁶⁴ Gv, Ma	ARPL ⁶⁴ Ew, Gw MOV ⁶⁴ _{SXD} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Eb, lb	Ev, lz	Eb, lb ⁶⁴	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOV ⁶⁴ _{S/B} Yb, Xb	MOV ⁶⁴ _{S/W/D/Q} Yv, Xv	CMPS ⁶⁴ _B Xb, Yb	CMPS ⁶⁴ _{W/D} Xv, Yv
B	AL/R8B, lb	CL/R9B, lb	DL/R10B, lb	BL/R11B, lb	AH/R12B, lb	CH/R13B, lb	DH/R14B, lb	BH/R15B, lb
C	Eb, lb	Ev, lb	near RET ⁶⁴ lw	near RET ⁶⁴	LES ⁶⁴ Gz, Mp VEX+2byte	LDS ⁶⁴ Gz, Mp VEX+1byte	Grp 11 ^{1A} - MOV Eb, lb	Ev, lz
D	Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM ⁶⁴ lb	AAD ⁶⁴ lb		XLAT/ XLATB
E	LOOPNE ⁶⁴ / LOOPNZ ⁶⁴ Jb	LOOPE ⁶⁴ / LOOPZ ⁶⁴ Jb	LOOP ⁶⁴ Jb	Jrcxz ⁶⁴ / Jb	AL, lb	IN eAX, lb	lb, AL	OUT lb, eAX
F	LOCK (Prefix)	INT1	REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3 ^{1A} Eb	Ev

Table A-2. One-byte Opcode Map: (08H – FFH) *

	8	9	A	B	C	D	E	F
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH CS ⁱ⁶⁴	2-byte escape (Table A-3)
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH DS ⁱ⁶⁴	POP DS ⁱ⁶⁴
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=CS (Prefix)	DAS ⁱ⁶⁴
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=DS (Prefix)	AAS ⁱ⁶⁴
4	DEC ⁱ⁶⁴ general register / REX ^{o64} Prefixes							
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	POP ^{d64} into general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH ^{d64} lz	IMUL Gv, Ev, lz	PUSH ^{d64} lb	IMUL Gv, Ev, lb	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc ⁱ⁶⁴ , Jb- Short displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A ^{1A} POP ^{d64} Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL ⁱ⁶⁴ Ap	FWAIT/ WAIT	PUSHF/D/Q ^{d64} / Fv	POPF/D/Q ^{d64} / Fv	SAHF	LAHF
A	TEST AL, lb		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv
B	MOV immediate word or double into word, double, or quad register							
	rAX/r8, lv	rCX/r9, lv	rDX/r10, lv	rBX/r11, lv	rSP/r12, lv	rBP/r13, lv	rSI/r14, lv	rDI/r15, lv
C	ENTER lw, lb	LEAVE ^{d64}	far RET lw	far RET	INT3	INT lb	INTO ⁱ⁶⁴	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	near CALL ⁱ⁶⁴ Jz	near ^{f64} Jz	JMP far ⁱ⁶⁴ Ap	short ^{f64} Jb	AL, DX	eAX, DX	DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 ^{1A}	INC/DEC Grp 5 ^{1A}

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-3. Two-byte Opcode Map: 00H – 77H (First Byte is 0FH) *

	pxf	0	1	2	3	4	5	6	7
0		Grp 6 ^{1A}	Grp 7 ^{1A}	LAR Gv, Ew	LSL Gv, Ew		SYSCALL ⁰⁶⁴	CLTS	SYSRET ⁰⁶⁴
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps ^{v1} Vdq, Hq, Mq vmovhlps Vdq, Hq, Uq	vmovhps ^{v1} Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpd ^{v1} Vdq, Hq, Mq	vmovhpd ^{v1} Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy, Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12 ^{1A})	(Grp 13 ^{1A})	(Grp 14 ^{1A})	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper ^v vzeroall ^v
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshuffw Vx, Wx, Ib							
	F2	vpshufw Vx, Wx, Ib							

Table A-3. Two-byte Opcode Map: 08H – 7FH (First Byte is 0FH) *

	px	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2 ^{1B}		prefetchw(/1) Ev		
1		Prefetch ^{1C} (Grp 16 ^{1A})	Reserved-NOP	bndldx	bndstx	Reserved-NOP			NOP /0 Ev
	66			bndmov	bndmov				
	F3			bndcl	bndmk				
	F2			bndcu	bndcn				
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvtps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvtpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtss2ss Vss, Hss, Ey		vcvtss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtss2sd Vsd, Hsd, Ey		vcvtss2si Gy, Wsd	vcvtss2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvtss2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtss2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsbpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx, Vx
	F3							vmovq Vq, Wq	vmovdqu Wx, Vx
	F2					vhaddps Vps, Hps, Wps	vhsbps Vps, Hps, Wps		

Table A-3. Two-byte Opcode Map: 80H – F7H (First Byte is 0FH) *

	pxf	0	1	2	3	4	5	6	7
8		Jcc ⁶⁴ , Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH ^{d64} FS	POP ^{d64} FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb Ev, Gv		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb Gv, Ew	
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9 ^{1A}
	66			vcmpsd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpss Vss,Hss,Wss,Ib					
	F2			vcmpsdb Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvttd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

Table A-3. Two-byte Opcode Map: 88H – FFH (First Byte is 0FH) *

	pxf	8	9	A	B	C	D	E	F
8		Jcc ⁶⁴ , Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH ^{d64} GS	POP ^{d64} GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 ^{1A}) ^{1C}	IMUL Gv, Ev
B		JMPE (reserved for emulator on IPF)	Grp 10 ^{1A} Invalid Opcode ^{1B}	Grp 8 ^{1A} Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSB Gv, Eb	Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpwr Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	UD0
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-4. Three-byte Opcode Map: 00H – F7H (First Two Bytes are 0F 38H) *

	pxf	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phadd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphadd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtp2ps ^V Vx, Wx, lb	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps ^V Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd ^V Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q ^V Vx, Hx, Wx	vpsravd ^V Vx, Hx, Wx	vpsrlvd/q ^V Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q ^V Vx,Hx,Wx	vgatherqd/q ^V Vx,Hx,Wx	vgatherdps/d ^V Vx,Hx,Wx	vgatherqps/d ^V Vx,Hx,Wx			vfmaddsub132ps/d ^V Vx,Hx,Wx	vfmsubadd132ps/d ^V Vx,Hx,Wx
A	66							vfmaddsub213ps/d ^V Vx,Hx,Wx	vfmsubadd213ps/d ^V Vx,Hx,Wx
B	66							vfmaddsub231ps/d ^V Vx,Hx,Wx	vfmsubadd231ps/d ^V Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN ^V Gy, By, Ey	Grp 17 ^{1A}		BZHI ^V Gy, Ey, By		BEXTR ^V Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX ^V Gy, Ey, By
	F3						PEXT ^V Gy, By, Ey	ADOX Gy, Ey	SARX ^V Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP ^V Gy, By, Ey	MULX ^V By,Gy,rDX,Ey	SHRX ^V Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

Table A-4. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 38H) *

	pxf	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps ^v Vx,Hx,Wx	vpermilpd ^v Vx,Hx,Wx	vtestps ^v Vx, Wx	vtestpd ^v Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss ^v Vx, Wd	vbroadcastsd ^v Vqq, Wq	vbroadcastf128 ^v Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2		vpmuldq Vx, Hx, Wx	vpcmpqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackudsw Vx, Hx, Wx	vmaskmovps ^v Vx,Hx,Mx	vmaskmovpd ^v Vx,Hx,Mx	vmaskmovps ^v Mx,Hx,Vx	vmaskmovpd ^v Mx,Hx,Vx
	66	vpmins b Vx, Hx, Wx	vpmins d Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
3									
	66	vpbroadcast ^v Vx, Wx	vpbroadcastq ^v Vx, Wx	vpbroadcasti128 ^v Vqq, Mdq					
4									
	66	vpbroadcastb ^v Vx, Wx	vpbroadcastw ^v Vx, Wx						
5									
	66					vpmaskmovd/q ^v Vx,Hx,Mx		vpmaskmovd/q ^v Mx,Vx,Hx	
6									
	66	vfmadd132ps/d ^v Vx, Hx, Wx	vfmadd132ss/d ^v Vx, Hx, Wx	vfmsub132ps/d ^v Vx, Hx, Wx	vfmsub132ss/d ^v Vx, Hx, Wx	vfnmadd132ps/d ^v Vx, Hx, Wx	vfnmadd132ss/d ^v Vx, Hx, Wx	vfmsub132ps/d ^v Vx, Hx, Wx	vfmsub132ss/d ^v Vx, Hx, Wx
7									
	66	vfmadd213ps/d ^v Vx, Hx, Wx	vfmadd213ss/d ^v Vx, Hx, Wx	vfmsub213ps/d ^v Vx, Hx, Wx	vfmsub213ss/d ^v Vx, Hx, Wx	vfnmadd213ps/d ^v Vx, Hx, Wx	vfnmadd213ss/d ^v Vx, Hx, Wx	vfmsub213ps/d ^v Vx, Hx, Wx	vfmsub213ss/d ^v Vx, Hx, Wx
8									
	66	vfmadd231ps/d ^v Vx, Hx, Wx	vfmadd231ss/d ^v Vx, Hx, Wx	vfmsub231ps/d ^v Vx, Hx, Wx	vfmsub231ss/d ^v Vx, Hx, Wx	vfnmadd231ps/d ^v Vx, Hx, Wx	vfnmadd231ss/d ^v Vx, Hx, Wx	vfmsub231ps/d ^v Vx, Hx, Wx	vfmsub231ss/d ^v Vx, Hx, Wx
9									
	66	sha1nexte Vdq,Wdq	sha1msg1 Vdq,Wdq	sha1msg2 Vdq,Wdq	sha256rmds2 Vdq,Wdq	sha256msg1 Vdq,Wdq	sha256msg2 Vdq,Wdq		
A									
	66				VAESIMC Vdq, Wdq	VAESEC Vdq,Hdq,Wdq	VAESENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
B									
	66								
C									
	66								
D									
	66								
E									
	66								
F									
	66 & F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

	px	0	1	2	3	4	5	6	7
0	66	vpermq ^v Vqq, Wqq, lb	vpermpd ^v Vqq, Wqq, lb	vblendd ^v Vx,Hx,Wx,lb		vpermilps ^y Vx, Wx, lb	vpermilpd ^y Vx, Wx, lb	vperm2f128 ^y Vqq,Hqq,Wqq,lb	
1	66					vpextrb Rd/Mb, Vdq, lb	vpextrw Rd/Mw, Vdq, lb	vpextrd/q Ey, Vdq, lb	vextractps Ed, Vdq, lb
2	66	vpinsrb Vdq,Hdq,Ry/Mb,lb	vinsertps Vdq,Hdq,Udq/Md,lb	vpinsrd/q Vdq,Hdq,Ey,lb					
3									
4	66	vdpps Vx,Hx,Wx,lb	vdppd Vdq,Hdq,Wdq,lb	vmpsadbw Vx,Hx,Wx,lb		vpcimulqdq Vdq,Hdq,Wdq,lb		vperm2i128 ^y Vqq,Hqq,Wqq,lb	
5									
6	66	vpcmpestrm Vdq, Wdq, lb	vpcmpestri Vdq, Wdq, lb	vpcmpistrm Vdq, Wdq, lb	vpcmpistri Vdq, Wdq, lb				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX ^v Gy, Ey, lb							

Table A-5. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 3AH) *

	px	8	9	A	B	C	D	E	F
0									palignr Pq, Qq, Ib
	66	vroundps Vx, Wx, Ib	vroundpd Vx, Wx, Ib	vroundss Vss, Wss, Ib	vroundsd Vsd, Wsd, Ib	vblendps Vx, Hx, Wx, Ib	vblendpd Vx, Hx, Wx, Ib	vpblendw Vx, Hx, Wx, Ib	vpaddingr Vx, Hx, Wx, Ib
1	66	vinserf128 ^v Vqq, Hqq, Wqq, Ib	vextractf128 ^v Wdq, Vqq, Ib				vcvtps2ph ^y Wx, Vx, Ib		
2									
3	66	vinserf128 ^v Vqq, Hqq, Wqq, Ib	vextractf128 ^v Wdq, Vqq, Ib						
4	66			vblendvps ^v Vx, Hx, Wx, Lx	vblendvpd ^v Vx, Hx, Wx, Lx	vpblendvb ^v Vx, Hx, Wx, Lx			
5									
6									
7									
8									
9									
A									
B									
C						sha1rmds4 Vdq, Wdq, Ib			
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

mod	nnn	R/M
-----	-----	-----

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

A.4.2 Opcode Extension Tables

See Table A-6 below.

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)								
				000	001	010	011	100	101	110	111	
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP	
8F	1A	mem, 11B		POP								
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR	
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX	
FE	4	mem, 11B		INC Eb	DEC Eb							
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL ^{f64} Ev	far CALL Ep	near JMP ^{f64} Ev	far JMP Mp	PUSH ^{d64} Ev		
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew			
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb	
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001)					SWAPGS ⁰⁶⁴ (000) RDTSCP (001)	
0F BA	8	mem, 11B						BT	BTS	BTR	BTC	
0F C7	9	mem			CMPXCH8B Mq					VMPTRLD Mq	VMPTRST Mq	
			66							VMCLEAR Mq		
		11B	F3								VMXON Mq	
			F3								RDRAND Rv	RDSEED Rv
0F B9	10	mem										
		11B										
C6	11	mem		MOV Eb, lb								
		11B									XABORT (000) lb	
C7	11	mem		MOV Ev, lz								
		11B									XBEGIN (000) Jz	
0F 71	12	mem										
		11B				psrlw Nq, lb		psraw Nq, lb		psllw Nq, lb		
		66				vpsrlw Hx,Ux,lb		vpsraw Hx,Ux,lb		vpsllw Hx,Ux,lb		
0F 72	13	mem										
		11B				psrld Nq, lb		psrad Nq, lb		pslld Nq, lb		
		66				vpsrld Hx,Ux,lb		vpsrad Hx,Ux,lb		vpslld Hx,Ux,lb		
0F 73	14	mem										
		11B				psrlq Nq, lb				psllq Nq, lb		
		66				vpsrlq Hx,Ux,lb	vpsrldq Hx,Ux,lb			vpsllq Hx,Ux,lb	vpslldq Hx,Ux,lb	

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B	F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry		lfence	mfence	sfence
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2	Reserved NOP			
		11B		Reserved NOP							
VEX.0F38 F3	17	mem			BLSR ^v By, Ey	BLSMSK ^v By, Ey	BLSI ^v By, Ey				
		11B									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	EXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMPP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

OPCODE MAP

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

This page was
intentionally left
blank.

13. Updates to Chapter 2, Volume 3A

Change bars and green text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Added details on protection keys for supervisor-mode pages, Control-flow Enforcement Technology (CET) details, and 5-level paging details where applicable. Typo corrections as needed.

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.

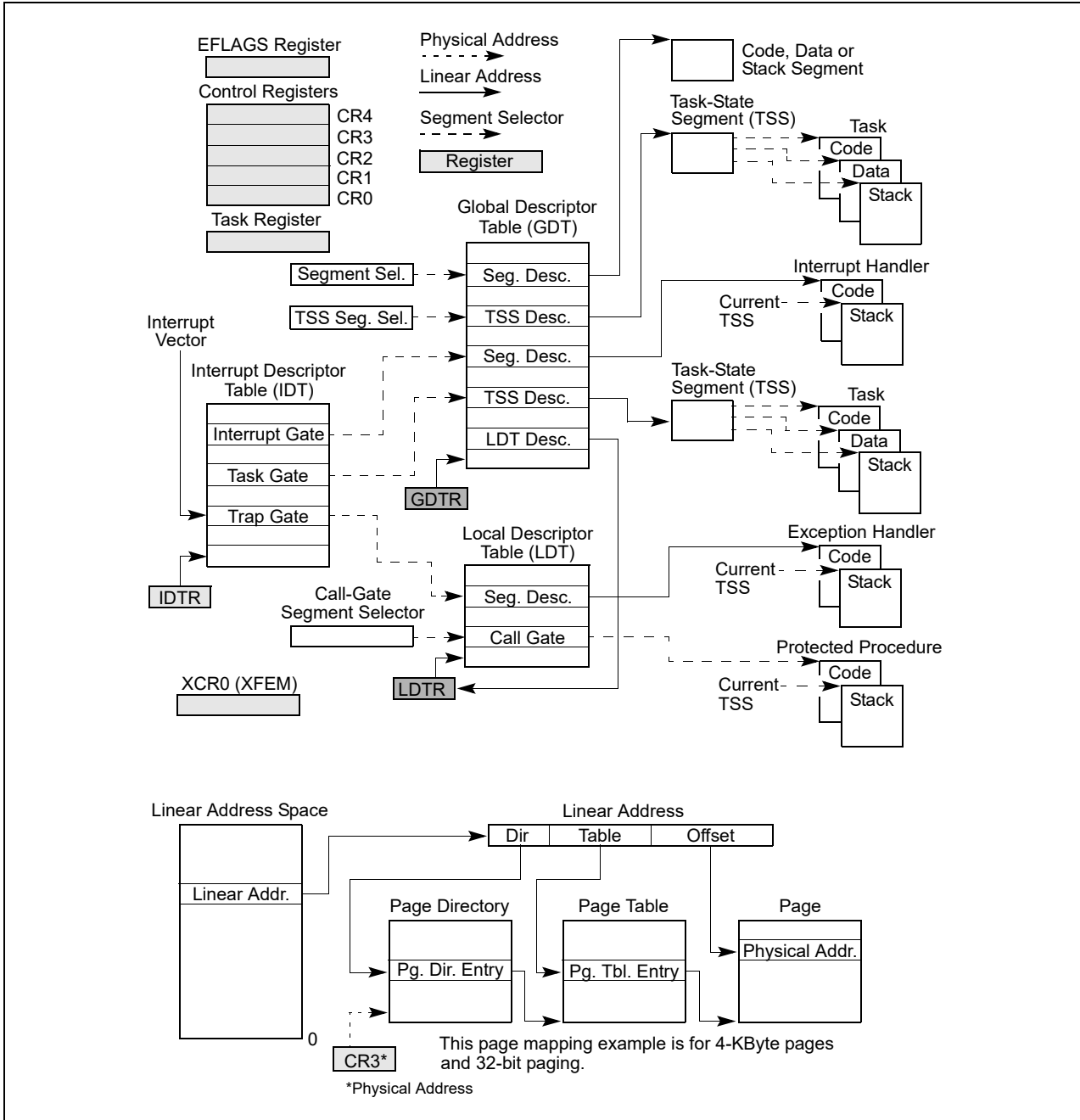


Figure 2-1. IA-32 System-Level Registers and Data Structures

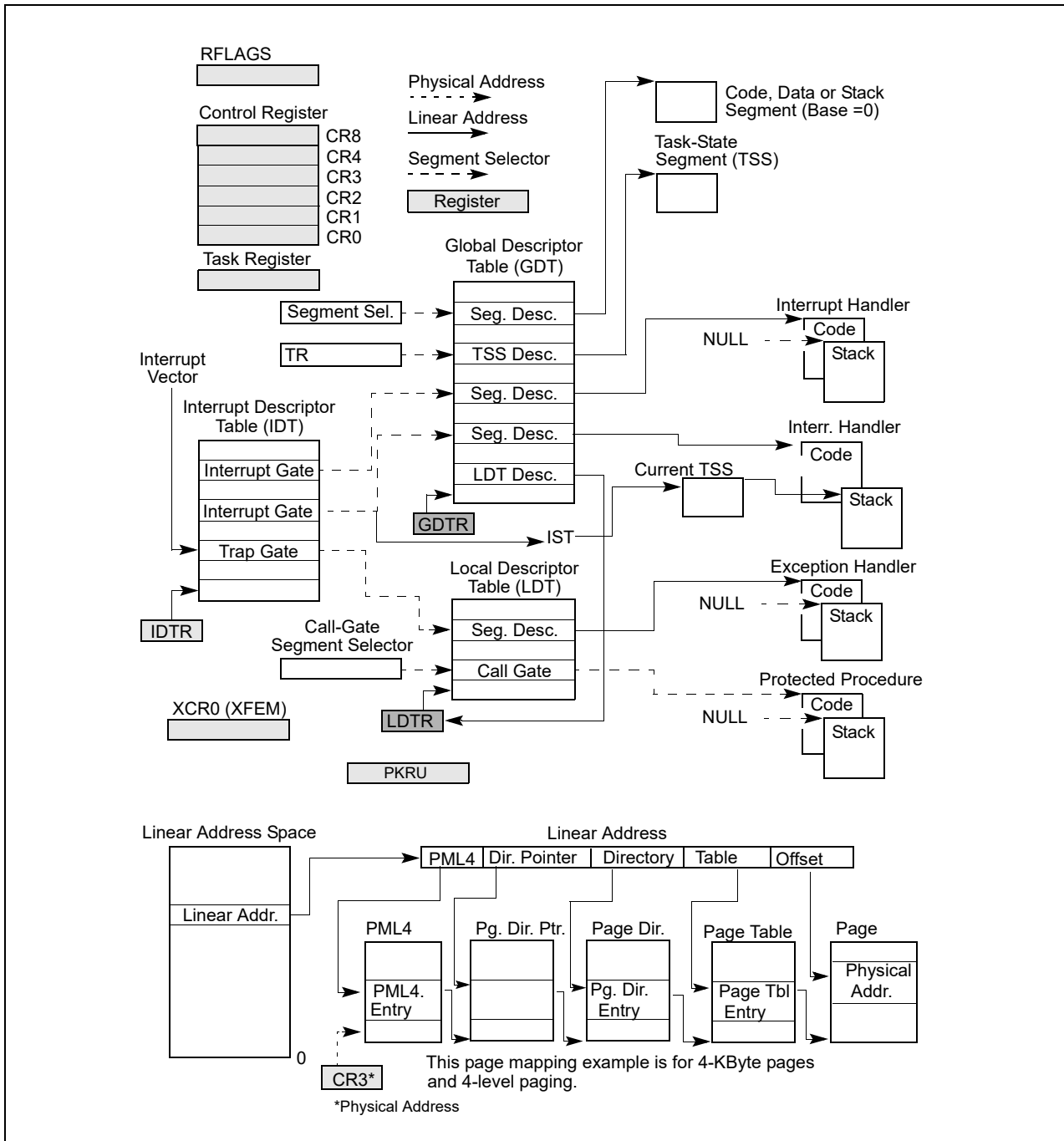


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

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 procedure¹ 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 *n*, INTO, INT3, INT1, 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 both compatibility mode and 64-bit mode, four or five levels of system data structures are used (see Chapter 4, "Paging"). These include the following:

- **The page map level 5 (PML5)** — An entry in the PML5 table contains the physical address of the base of a PML4 table, access rights, and memory management information. The base physical address of the PML5 table is stored in CR3. The PML5 table is used only with 5-level paging.
- **A 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. With 4-level paging, there is only one PML4 table and its base physical address 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 Intel® Resource Director Technology (Intel® RDT) 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.

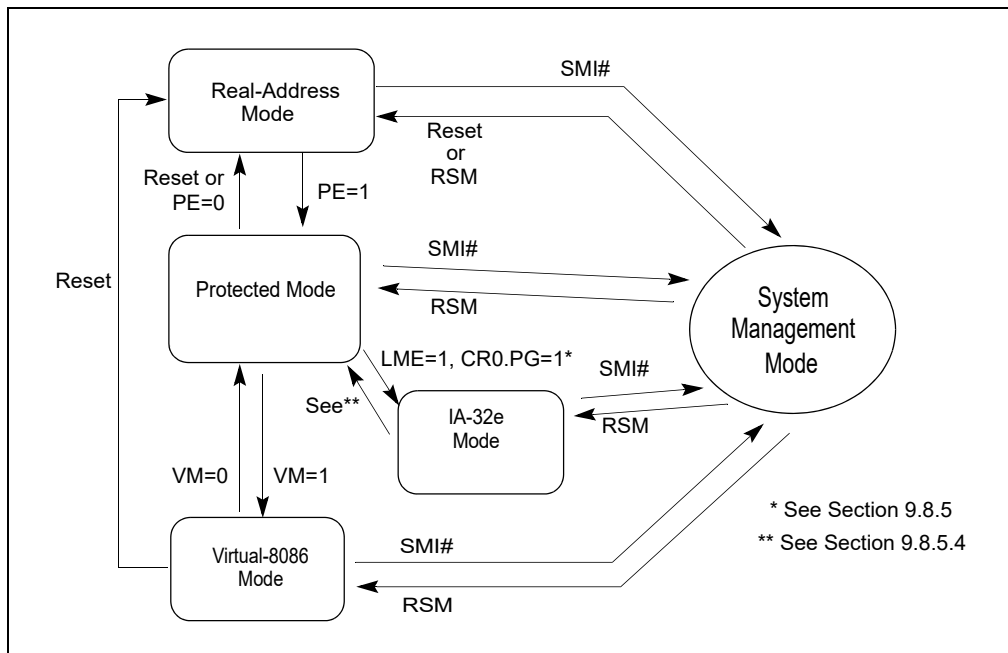


Figure 2-3. Transitions Among the Processor’s Operating Modes

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.

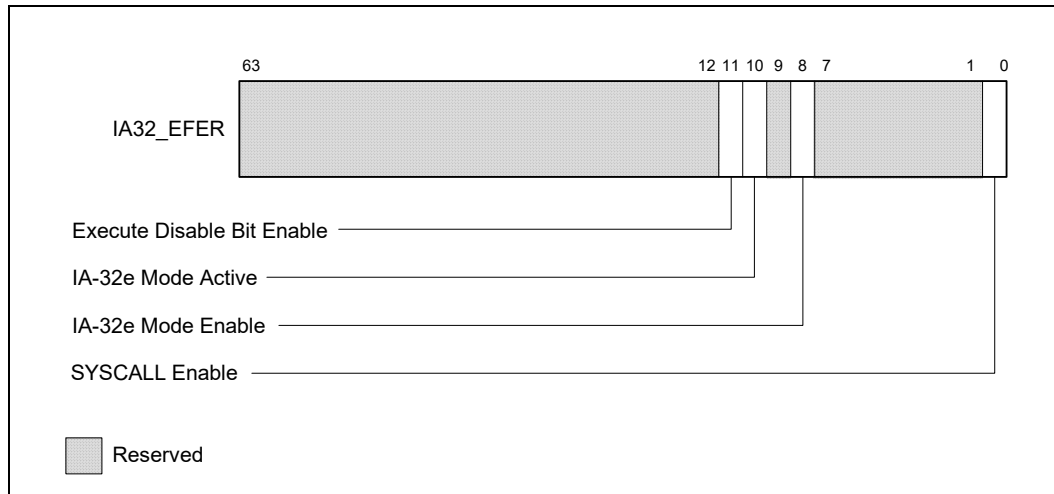


Figure 2-4. IA32_EFER MSR Layout

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.

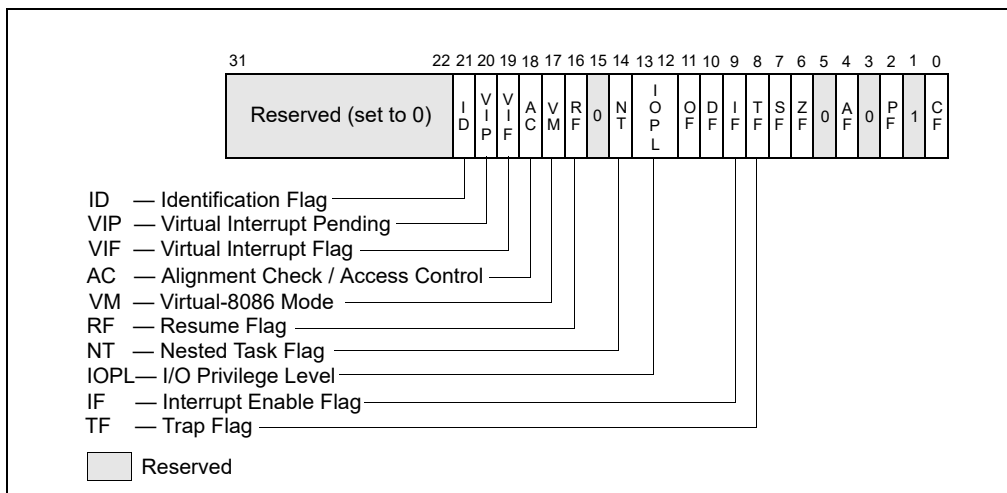


Figure 2-5. System Flags in the EFLAGS Register

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 19, “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.

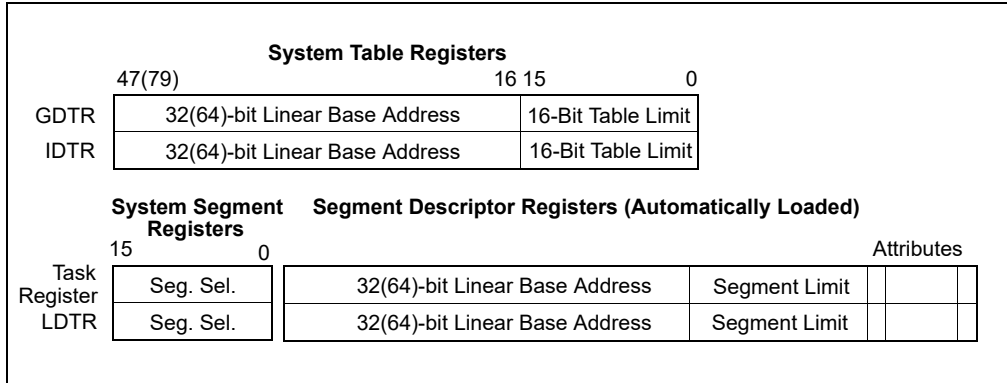


Figure 2-6. Memory Management Registers

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 of CR3 in the range 51:12 that are beyond the processor’s physical-address width are reserved and must be 0.
- The MOV CR2 instruction does not check that address written to CR2 is canonical.
- 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. With 4-level paging and 5-level paging, the CR3 register contains the base address of the PML4 table and PML5 table, respectively. If PCIDs are enabled, CR3 has a format different from that illustrated in Figure 2-7. See Section 4.5, “4-Level Paging and 5-Level Paging.”

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.

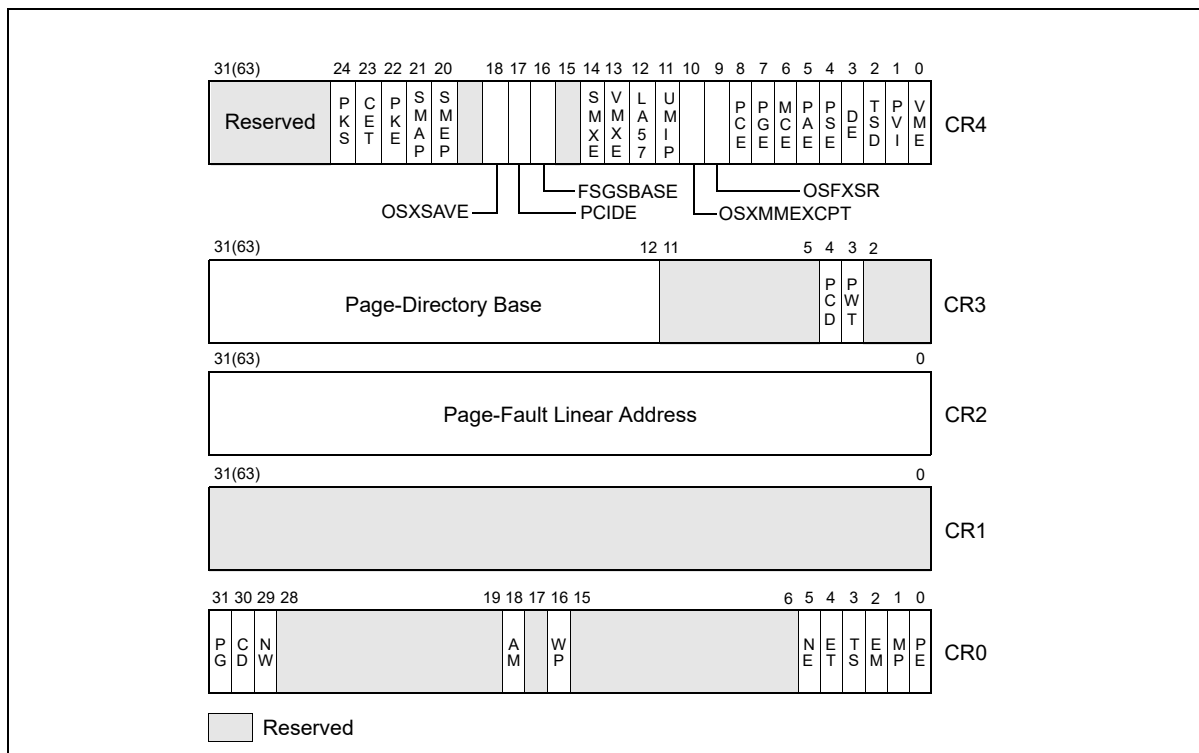


Figure 2-7. Control Registers

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

CR0.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.

CR0.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."

CR0.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.

CR0.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.

CR0.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. **This flag must be set before software can set CR4.CET, and it cannot be cleared as long as CR4.CET = 1 (see below).**

CR0.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*.

CR0.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.

CR0.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, PREFETCHh, 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

CRO 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.

CR0.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, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

CR0.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.

CR0.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."

CR3.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 paging² or 5-level paging if CR4.PCIDE=1.

CR3.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 or 5-level paging if CR4.PCIDE=1.

CR4.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-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.”

CR4.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.”

CR4.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).

CR4.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.”

CR4.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.”

CR4.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.”

CR4.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.”

CR4.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-pointer-table entry](#), a [page-directory entry](#), or a [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.”

2. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

CR4.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.

CR4.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, PREFETCHh, 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, PREFETCHh, 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.

CR4.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.

CR4.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).

CR4.LA57

57-bit linear addresses (bit 12 of CR4) — When set in IA-32e mode, the processor uses 5-level paging to translate 57-bit linear addresses. When clear in IA-32e mode, the processor uses 4-level paging to translate 48-bit linear addresses. This bit cannot be modified in IA-32e mode.

See also: Chapter 4, "Paging."

CR4.VMXE

VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set. See Chapter 23, "Introduction to Virtual Machine Extensions."

CR4.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*.

CR4.FSGSBASE

FSGSBASE-Enable Bit (bit 16 of CR4) — Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.

CR4.PCID

PCID-Enable Bit (bit 17 of CR4) — Enables process-context identifiers (PCIDs) when set. See Section 4.10.1, "Process-Context Identifiers (PCIDs)". Applies only in IA-32e mode (if IA32_EFER.LMA = 1).

CR4.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”.

CR4.SMEP

SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, “Access Rights”.

CR4.SMAP

SMAP-Enable Bit (bit 21 of CR4) — Enables supervisor-mode access prevention (SMAP) when set. See Section 4.6, “Access Rights”.

CR4.PKE

Enable protection keys for user-mode pages (bit 22 of CR4) — 4-level paging and 5-level paging associate each user-mode linear address with a protection key. When set, this flag indicates (via CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4]) that the operating system supports use of the PKRU register to specify, 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.

CR4.CET

Control-flow Enforcement Technology (bit 23 of CR4) — Enables control-flow enforcement technology when set. See Chapter 18, “Control-flow Enforcement Technology (CET)” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*. This flag can be set only if CR0.WP is set, and it must be clear before CR0.WP can be cleared (see below).

CR4.PKS

Enable protection keys for supervisor-mode pages (bit 24 of CR4) — 4-level paging and 5-level paging associate each supervisor-mode linear address with a protection key. When set, this flag allows use of the IA32_PKRS MSR to specify, for each protection key, whether supervisor-mode linear addresses with that protection key can be read or written.

CR8.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.

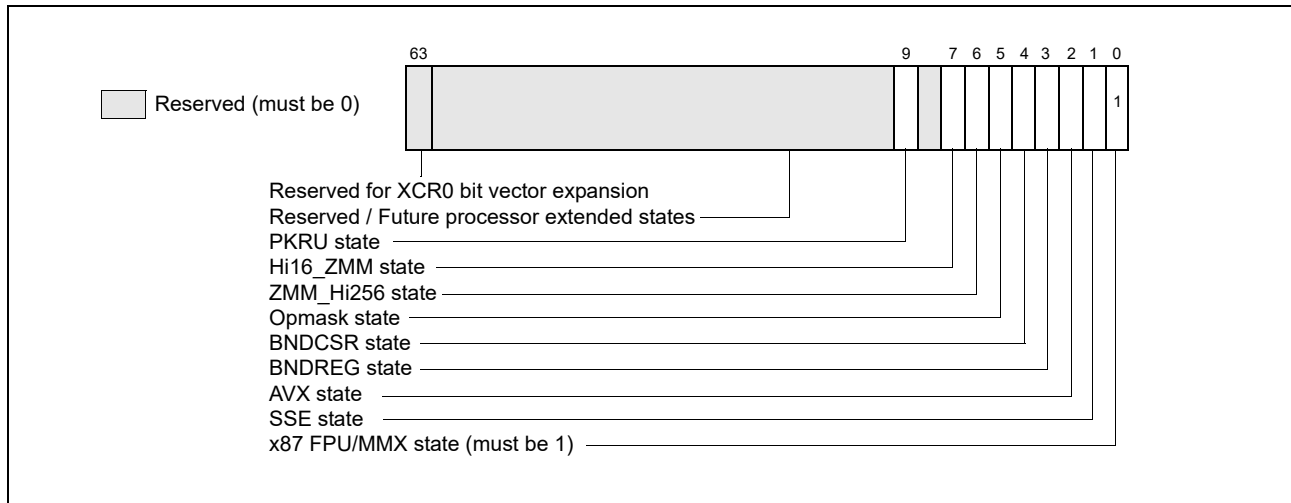


Figure 2-8. XCR0

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 REGISTERS (PKRU AND IA32_PKRS)

Processors may support either or both of two protection-key rights registers: PKRU for user-mode pages and the IA32_PKRS MSR (MSR index 6E1H) for supervisor-mode pages. 4-level paging and 5-level paging associate a 4-bit **protection key** with each page. The protection-key rights registers determine accessibility based on a page's protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for **user-mode pages**. When CR4.PKE = 1, software can use the **protection-key rights register for user pages** (PKRU) to specify the access rights for user-mode pages for each protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, the processor supports the protection-key feature for **supervisor-mode pages**. When CR4.PKS = 1, software can use the **protection-key rights register for supervisor pages** (the IA32_PKRS MSR) to specify the access rights for supervisor-mode pages for each protection key.

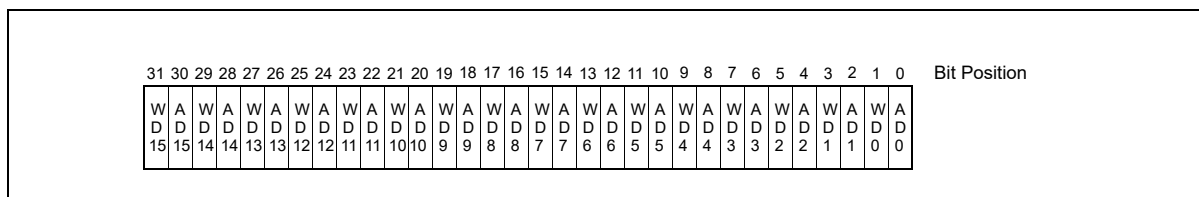


Figure 2-9. Format of Protection-Key Rights Registers

The format of each protection-key rights register is given in Figure 2-9. Each contains 16 pairs of disable controls to prevent data accesses to linear addresses (**user-mode or supervisor-mode, depending on the register**) based on their protection keys. Each protection key i ($0 \leq i \leq 15$) is associated with two bits in each **protection-key rights register**:

- Bit $2i$, shown as "AD i " (access disable): if set, the processor prevents any data accesses to linear addresses (**user-mode or supervisor-mode, depending on the register**) with protection key i .
- Bit $2i+1$, shown as "WD i " (write disable): if set, the processor prevents write accesses to linear addresses (**user-mode or supervisor-mode, depending on the register**) with protection key i .

(Bits 63:32 of the IA32_PKRS MSR are reserved and must be zero.)

See Section 4.6.2, "Protection Keys," for details of how the processor uses the **protection-key rights registers** to control accesses to linear addresses.

Software can read and write PKRU using the RDPKRU and WRPKRU instructions. The IA32_PKRS MSR can be read and written with the RDMSR and WRMSR instructions. Writes to the IA32_PKRS MSR using WRMSR are not serializing.

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 CRO	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 XCRO	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.

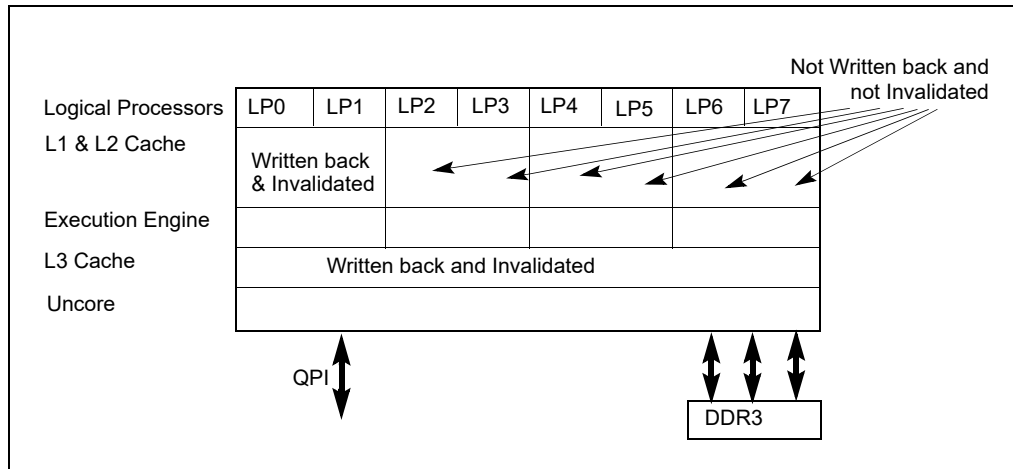


Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy

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 $\sim 9.5 \times 10^{16}$ 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 17.17, “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).

14. Updates to Chapter 4, Volume 3A

Change bars and green text 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: Added details on protection keys for supervisor-mode pages, and 5-level paging details where applicable. Typo corrections as needed.

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 **four** 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 four 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, **LA57**, PCIDE, SMEP, SMAP, PKE, CET, and **PKS** flags in control register CR4 (bit 4, bit 5, bit 7, **bit 12**, bit 17, bit 20, bit 21, bit 22, bit 23, and **bit 24**, 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 **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, **CR4.LA57**, and IA32_EFER.LME determine whether paging is **enabled** and, if so, which of **four** 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, CR4.CET, **CR4.PKS**, and IA32_EFER.NXE modify the operation of the different paging modes.

4.1.1 Four 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, **CR4.LA57**, 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. (CR4.CET is also ignored insofar as it affects linear-address access rights.)

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 **four** paging modes is used. The values of CR4.PAE, **CR4.LA57**, and IA32_EFER.LME determine which paging mode is used:

- If 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, CR4.SMAP, and CR4.CET as described in Section 4.1.3 and Section 4.6.

- If 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, CR4.CET, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.
- If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 0, **4-level paging**¹ is used.² 4-level paging is detailed in Section 4.5 (along with 5-level paging). 4-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.
- If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 1, **5-level paging** is used. 5-level paging is detailed in Section 4.5 (along with 4-level paging). 5-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.

NOTE

32-bit paging and PAE paging can be used only in legacy protected mode (IA32_EFER.LME = 0). In contrast, 4-level paging and 5-level paging can be used only IA-32e mode (IA32_EFER.LME = 1).

The four 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 and 5-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 and 5-level paging, each linear address is associated with a **protection key**. Software can use the protection-key rights registers to disable, for each protection key, how certain accesses to linear addresses associated with that protection key.

Table 4-1 illustrates the principal differences between the four paging modes.

Table 4-1. Properties of Different Paging Modes

Paging Mode	PG in CRO	PAE in CR4	LME in IA32_EFER	LA57 in CR4	Lin.-Addr. Width	Phys.-Addr. Width ¹	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
None	0	N/A	N/A	N/A	32	32	N/A	No	No
32-bit	1	0	0 ²	N/A	32	Up to 40 ³	4 KB 4 MB ⁴	No	No
PAE	1	1	0	N/A	32	Up to 52	4 KB 2 MB	Yes ⁵	No
4-level	1	1	1	0	48	Up to 52	4 KB 2 MB 1 GB ⁶	Yes ⁵	Yes ⁷
5-level	1	1	1	1	57	Up to 52	4 KB 2 MB 1 GB ⁶	Yes ⁵	Yes ⁷

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 uses either 4-level paging or 5-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.

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. 32-bit paging uses 4-MByte pages 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. Processors that support 4-level paging or 5-level paging do not necessarily 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.

Because 32-bit paging and PAE paging are used only in legacy protected mode and because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

4-level paging and 5-level paging are used only in IA-32e mode. IA-32e mode has two sub-modes:

- Compatibility mode. This sub-mode uses only 32-bit linear addresses. In this sub-mode, 4-level paging and 5-level paging treat bits 63:32 of such an address as all 0.
- 64-bit mode. While this sub-mode produces 64-bit linear addresses, the processor enforces **canonicity**, meaning that the upper bits of such an address are identical: bits 63:47 for 4-level paging and bits 63:56 for 5-level paging. 4-level paging (respectively, 5-level paging) does not use bits 63:48 (respectively, bits 63:57) of such addresses.

4.1.2 Paging-Mode Enabling

If CR0.PG = 1, a logical processor is in one of four paging modes, depending on the values of CR4.PAE, IA32_EFER.LME, and CR4.LA57. Figure 4-1 illustrates how software can enable these modes and make transitions between them. The following items identify certain limitations and other details:

- 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)).
- 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)).
- One node in Figure 4-1 is labeled "IA-32e mode." This node represents either 4-level paging (if CR4.LA57 = 0) or 5-level paging (if CR4.LA57 = 1). As noted in the following items, software cannot modify CR4.LA57 (effecting transition between 4-level paging and 5-level paging) without first disabling paging.
- CR4.PAE and CR4.LA57 cannot be modified while either 4-level paging or 5-level paging is in use (when 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 transition between 32-bit paging and PAE paging by changing the value of CR4.PAE with MOV to CR4.
- Software cannot transition directly between 4-level paging (or 5-level paging) and any of other paging mode. It must first disable paging (by clearing CR0.PG with MOV to CR0), then set CR4.PAE, IA32_EFER.LME, and CR4.LA57 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 modify CR4.PAE, IA32_EFER.LME, or CR4.LA57 while 4-level paging or 5-level paging is enabled causes 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.

1. If the logical processor is in 64-bit mode or if CR4.PCIDE = 1, an attempt to clear CR0.PG causes a general-protection exception (#GP). Software should transition to compatibility mode and clear CR4.PCIDE before attempting to disable paging.

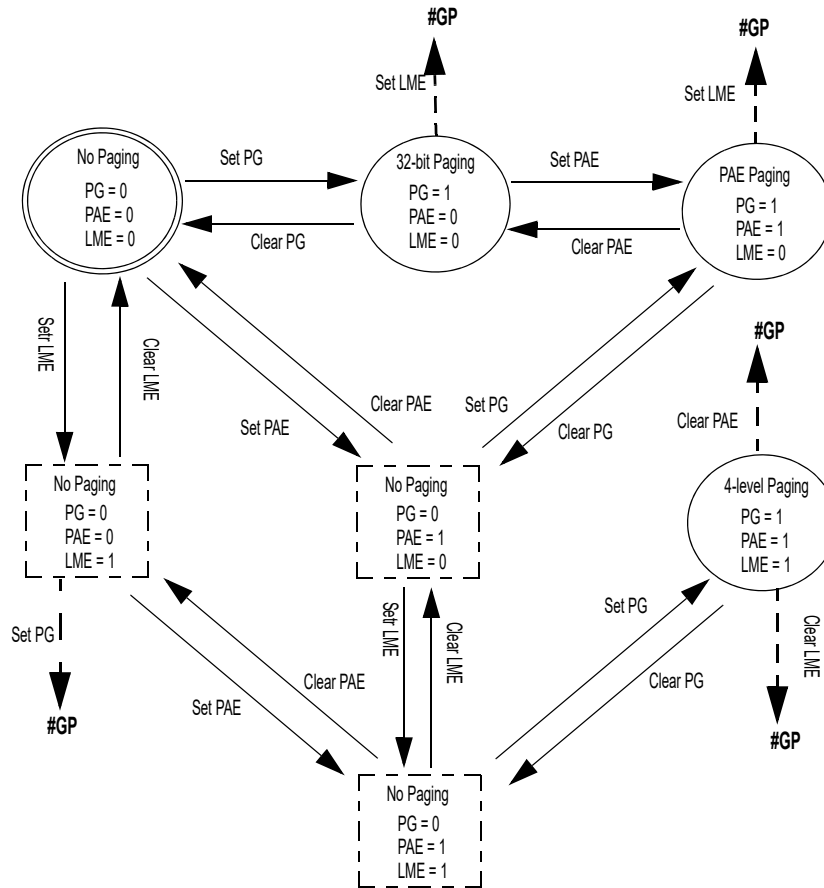


Figure 4-1. Enabling and Changing Paging Modes

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, PKE, CET, and PKS flags in CR4 (bit 4, bit 7, bit 17, bit 20, bit 21, bit 22, bit 23, and bit 24, 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, 4-level paging, and 5-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 and 5-level paging. 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 and CR4.PKS enable specification of access rights based on **protection keys**. 4-level paging and 5-level paging associate each linear address with a protection key. When CR4.PKE = 1, the PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written. When CR4.PKS = 1, the IA32_PKRS MSR does the same for supervisor-mode linear addresses. See Section 4.6 for more information.

CR4.CET enables **control-flow enforcement technology**, including the shadow-stack feature. If CR4.CET = 1, certain memory accesses are identified as **shadow-stack accesses** and certain linear addresses translate to **shadow-stack pages**. Section 4.6 explains how access rights are determined for these accesses and pages. (The processor allows CR4.CET to be set only if CR0.WP is also set.)

IA32_EFER.NXE enables execute-disable access rights for PAE paging, 4-level paging, and 5-level paging. If IA32_EFER.NXE = 1, instruction fetches can be prevented from specified linear addresses (even if data reads from the addresses are 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 PAE paging, 4-level paging, or 5-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 and 5-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 for user-mode pages.**
If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, CR4.PKE may be set to 1, enabling protection keys for user-mode pages (see Section 4.6).
- **OSPKE: enabling of protection keys for user-mode pages.**
CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4] returns the value of CR4.PKE. Thus, protection keys for user-mode pages are enabled if this flag is 1 (see Section 4.6).
- **CET: control-flow enforcement technology.**
If CPUID.(EAX=07H,ECX=0H):ECX.CET_SS [bit 7] = 1, CR4.CET may be set to 1, enabling shadow-stack pages (see Section 4.6).
- **LA57: 57-bit linear addresses and 5-level paging.**
If CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 1, CR4.LA57 may be set to 1, enabling 5-level paging.
- **PKS: protection keys for supervisor-mode pages.**
If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, CR4.PKS may be set to 1, enabling protection keys for supervisor-mode pages (see Section 4.6).
- **NX: execute disable.**
If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing software 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 may be supported with 4-level paging and 5-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 IA-32e mode (with either 4-level paging or 5-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 reported as follows:
 - If CPUID.80000001H:EDX.LM [bit 29] = 0, the value is reported as 32.
 - If CPUID.80000001H:EDX.LM [bit 29] = 1 and CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 0, the value is reported as 48.
 - If CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 1, the value is reported as 57.
 (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)

4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All four paging modes translate linear addresses using **hierarchical paging structures**. This section provides an overview of their operation. Section 4.3, Section 4.4, Section 4.5, and Section 4.6 provide details for the four 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 the other paging modes, 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 four 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.)
- 5-level paging is similar to 4-level paging except that 5-level paging translates 57-bit linear addresses. Bits 56:48 identify the first paging-structure entry, while the remaining bits are used as with 4-level paging.

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 can use 4-MByte pages if CR4.PSE = 1. The other paging modes can use 2-MByte pages (regardless of the value of CR4.PSE). 4-level paging and 5-level paging can use 1-GByte pages if the processor supports them (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
PML5 table	PML5E	32-bit, PAE, 4-level	N/A		
		5-level	CR3	56:48	N/A (PS must be 0)
PML4 table	PML4E	32-bit, PAE	N/A		
		4-level	CR3	47:39	N/A (PS must be 0)
		5-level	PML5E		
Page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A (PS must be 0)
		4-level, 5-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, 5-level	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-KByte page
		PAE, 4-level, 5-level		20:12	

NOTES:

1. Not all processors support 1-GByte pages; see Section 4.1.4.
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 support 4-MByte pages with 32-bit paging; see Section 4.1.4.

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 as 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.
 - Bits 11:2 are bits 31:22 of the linear address.

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 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.

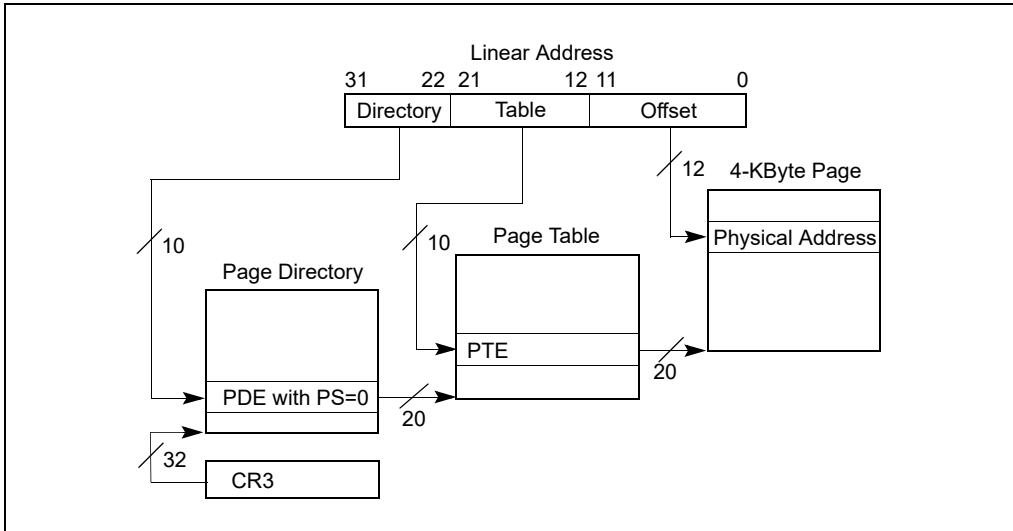


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

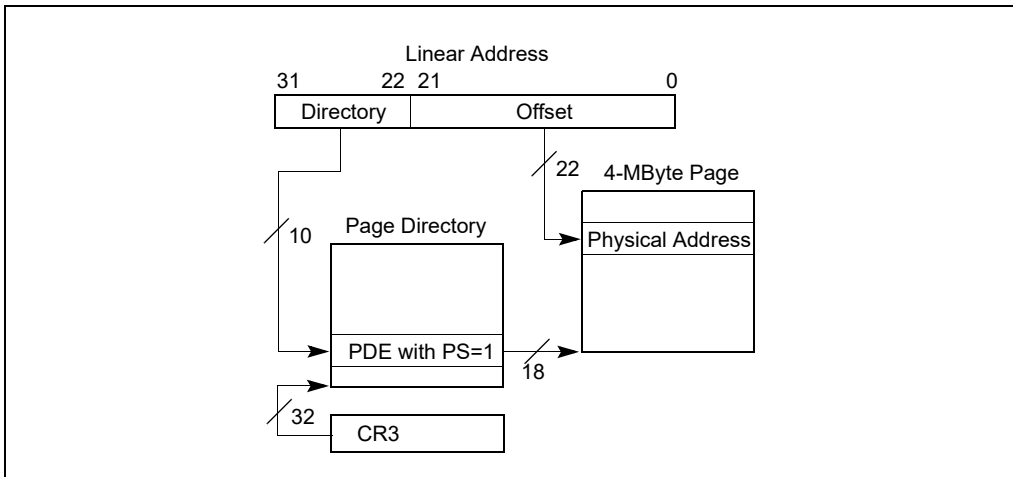


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

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.

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	
Address of page directory ¹												Ignored						PCD	PWT	Ignored				CR3								
Bits 31:22 of address of 4MB page frame						Reserved (must be 0)				Bits 39:32 of address ²		PAT	Ignored	G	<u>1</u>	D	A	PCD	PWT	U/S	R/W	<u>1</u>	PDE: 4MB page									
Address of page table												Ignored						<u>0</u>	I	g	n	A	PCD	PWT	U/S	R/W	<u>1</u>	PDE: page table				
Ignored												Ignored						<u>0</u>	PDE: not present													
Address of 4KB page frame												Ignored						G	P	A	T	D	A	PCD	PWT	U/S	R/W	<u>1</u>	PTE: 4KB page			
Ignored												Ignored						<u>0</u>	PTE: not present													

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 as 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.

PAGING

- 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.

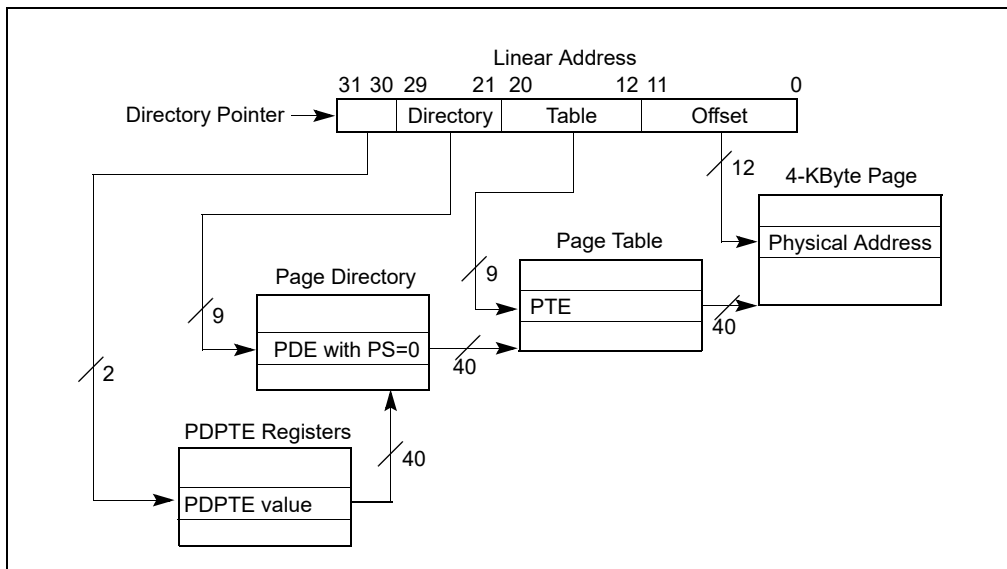


Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

1. See Section 4.1.4 for how to determine whether the PAT is supported.

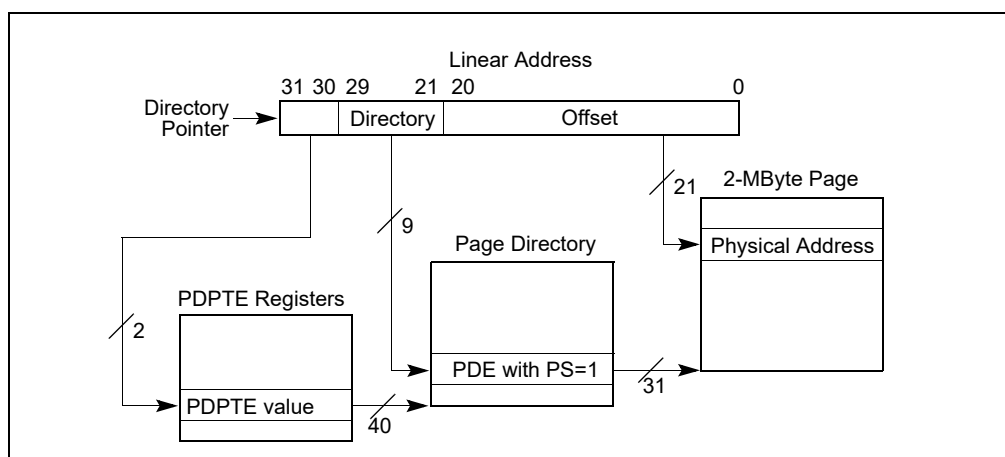


Figure 4-6. Linear-Address Translation to a 2-MByte Page using PAE Paging

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.

66	65	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	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
Ignored ²																Address of page-directory-pointer table																Ignored				CR3																														
Reserved ³																Address of page directory																Ign.	Rsvd.	P C D	P W T	R s v d	1	PDPTE: present																												
Ignored																Ignored																Ignored				0	PDPTE: not present																													
X D 4	Reserved																Address of 2MB page frame																Reserved	P A T	Ign.	G	1	D	A	P C D	P W T	R / S /	R /	1	PDE: 2MB page																					
X D 4	Reserved																Address of page table																Ign.	Ign.	0	A	P C D	P W T	R /	R /	1	PDE: page table																								
Ignored																Ignored																Ignored				0	PDE: not present																													
X D 4	Reserved																Address of 4KB page frame																Ign.	G	P A T	D	A	P C D	P W T	R /	R /	1	PTE: 4KB page																							
Ignored																Ignored																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 AND 5-LEVEL PAGING

Because the operation of 4-level paging and 5-level paging is very similar, they are described together in this section. The following items highlight the distinctions between the two paging modes:

- A logical processor uses 4-level paging if $CR0.PG = 1$, $CR4.PAE = 1$, $IA32_EFER.LME = 1$, and $CR4.LA57 = 0$. 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.
- A logical processor uses 5-level paging if $CR0.PG = 1$, $CR4.PAE = 1$, $IA32_EFER.LME = 1$, and $CR4.LA57 = 1$. 5-level paging translates 57-bit linear addresses to 52-bit physical addresses. Thus, 5-level paging supports a linear-address space sufficient to access the entire physical-address space.

Both paging modes translate linear addresses using a hierarchy of in-memory paging structures located using the contents of CR3, which is used to locate the first paging-structure. For 4-level paging, this is the PML4 table, and for 5-level paging it is the PML5 table. Use of CR3 with 4-level paging and 5-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 and 5-level paging if $CR4.PCIDE = 0$.

Table 4-12. Use of CR3 with 4-Level Paging and 5-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 or PML5 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 and 5-level paging if $CR4.PCIDE = 1$.

Table 4-13. Use of CR3 with 4-Level Paging and 5-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.

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.

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 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 and 5-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 for 4-level paging 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. (The process for 5-level paging is similar.)

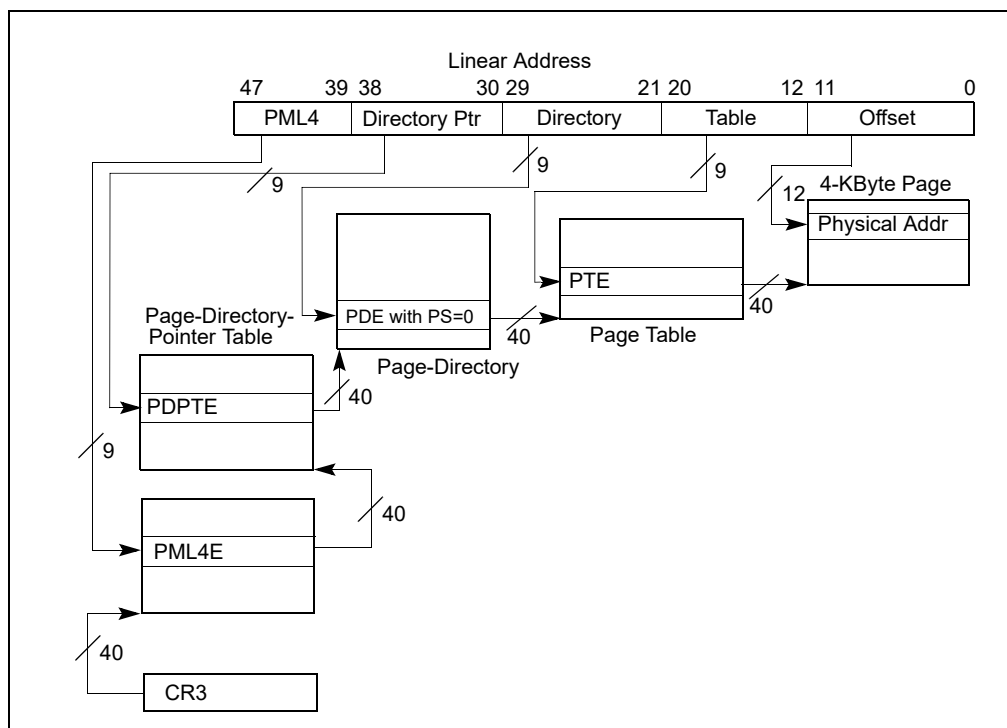


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

1. Not all processors support 1-GByte pages; see Section 4.1.4.

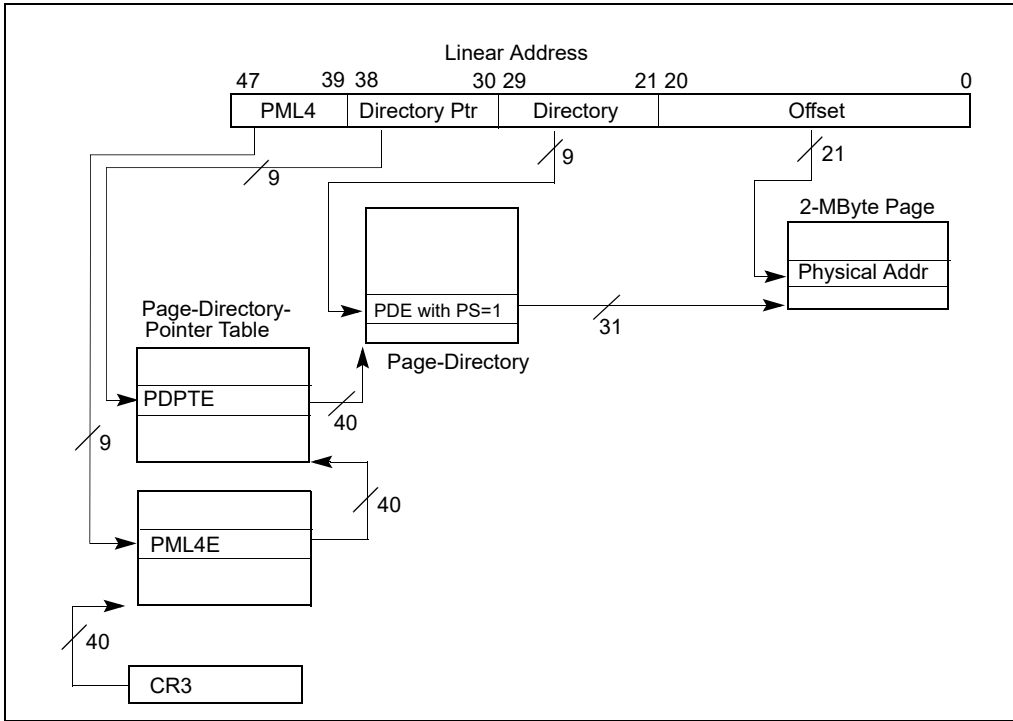


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

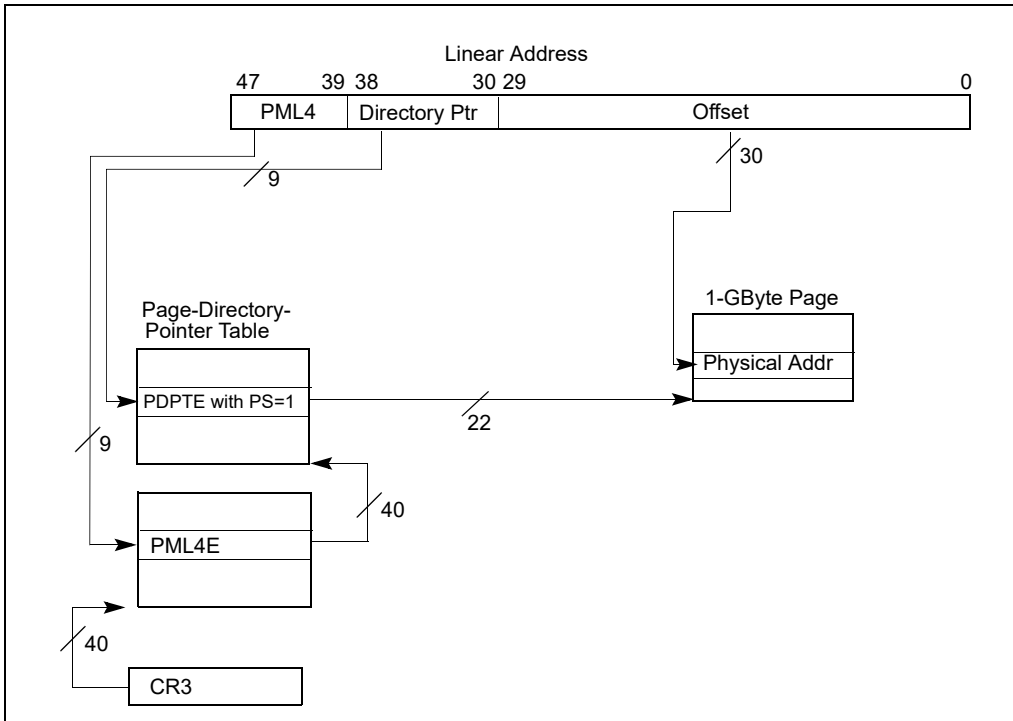


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

4-level paging and 5-level paging associate 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 remainder of this section describes the translation process used by 4-level paging and 5-level paging in more detail, as well as how the page size and protection key are determined. Because the process used by the two paging modes is similar, they are described together, with any differences identified, in the following items:

- With 5-level paging, a 4-KByte naturally aligned PML5 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). (4-level paging does not use a PML5 table and omits this step.) A PML5 table comprises 512 64-bit entries (PML5Es). A PML5E is selected using the physical address defined as follows:
 - Bits 51:12 are from CR3.
 - Bits 11:3 are bits 56:48 of the linear address.
 - Bits 2:0 are all 0.

Because a PML5E is identified using bits 56:48 of the linear address, it controls access to a 256-TByte region of the linear-address space.

- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (for 4-level paging; see Table 4-12) or in bits 51:12 of the PML4E (for 5-level paging; see Table 4-14). 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 (for 4-level paging) or in bits 51:12 of the PML4E (for 5-level paging).
 - 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 4-15). 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 4-16). The final physical address is computed as follows:
 - Bits 51:30 are from the PDPTe.
 - Bits 29:0 are from the original linear address.

The linear address's protection key is the value of bits 62:59 of the PDPTe (see Section 4.6.2).

- 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 4-17). 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 4-18). The final physical address is computed as follows:

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.

- Bits 51:21 are from the PDE.
- Bits 20:0 are from the original linear address.

The linear address’s protection key is the value of bits 62:59 of the PDE (see Section 4.6.2).

- 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 4-19). 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.
- Because a PTE is identified using bits 47:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-20). The final physical address is computed as follows:
 - Bits 51:12 are from the PTE.
 - Bits 11:0 are from the original linear address.

The linear address’s protection key is the value of bits 62:59 of the PTE (see Section 4.6.2).

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 in a paging-structure entry are reserved with 4-level paging and 5-level paging (assuming that the entry’s P flag is 1):

- Bits 51:MAXPHYADDR are reserved in every paging-structure entry.
- The PS flag is reserved in a PML5E or a PML4E.
- If 1-GByte pages are not supported, the PS flag is reserved in a PDPTE.¹
- If the PS flag in a PDPTE is 1, bits 29:13 of the entry are reserved.
- If the PS flag in a PDE is 1, bits 20:13 of the entry are reserved.
- If IA32_EFER.NXE = 0, the XD flag (bit 63) is reserved in every paging-structure entry.

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 and 5-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 PML5 Entry (PML5E) that References a PML4 Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a PML4 table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 256-TByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 256-TByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table referenced by this entry (see Section 4.9.2)

1. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.

Table 4-14. Format of a PML5 Entry (PML5E) that References a PML4 Table (Contd.)

Bit Position(s)	Contents
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 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
M-1:12	Physical address of 4-KByte aligned PML4 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 256-TByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-15. Format of a 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
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 4-16. Format of a 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 4-17)
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 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is not used to control access rights.
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 4-17. Format of a 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 4-16)
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 4-18. Format of a 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 4-19)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Table 4-18. Format of a 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 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is not used to control access rights.
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 4-19. Format of a 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 4-18)
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 4-20. Format of a 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 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is not used to control access rights.
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)

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.

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 5-level paging, and only if CR4.PKE = 1 or CR4.PKS = 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**.

When the shadow-stack feature of control-flow enforcement technology (CET) is enabled, certain accesses to linear addresses are considered **shadow-stack accesses** (see Section 18.2, “Shadow Stacks” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Like ordinary data accesses, each shadow-stack access is defined as being either a user access or a supervisor access. In general, a shadow-stack access is a user access if CPL = 3 and a supervisor access if CPL < 3. The WRUSS instruction is an exception; although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user accesses.

Shadow-stack accesses are allowed only to **shadow-stack addresses**. A linear address is a shadow-stack address if the following are true of the translation of the linear address: (1) the R/W flag (bit 1) is 0 and the dirty flag (bit 6) is 1 in the paging-structure entry that maps the page containing the linear address; and (2) the R/W flag is 1 in every other paging-structure entry controlling the translation of the linear address.

The following items detail how paging determines access rights (only the items noted explicitly apply to shadow-stack accesses):

- For supervisor-mode accesses:
 - Data may be read (implicitly or explicitly) from any supervisor-mode address with a protection key for which read access is permitted (see Section 4.6.2).
 - 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 (see Section 4.6.2).
 - 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 (see Section 4.6.2).
 - If EFLAGS.AC = 0 or the access is implicit, data may not be read from any user-mode address.

1. With PAE paging, the PDPTes do not determine access rights.

- 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 with a protection key for which write access is permitted (see Section 4.6.2).
 - 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 and with a protection key for which write access is permitted (see Section 4.6.2); 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 (see Section 4.6.2).
 - 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 (see Section 4.6.2).
 - 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 (see Section 4.6.2); 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 (see Section 4.6.2); 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.
- 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 other paging modes 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 other paging modes 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.
- Supervisor-mode shadow-stack accesses are allowed only to supervisor-mode shadow-stack addresses (see above).
- 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 (see Section 4.6.2).
 - 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 (see Section 4.6.2).
 - 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 other paging modes 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.
 - User-mode shadow-stack accesses made outside enclave mode are allowed only to user-mode shadow-stack addresses (see above). User-mode shadow-stack accesses made in enclave mode are treated like ordinary data accesses (see above).

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

4-level paging and 5-level paging associate a 4-bit protection key with each linear address (the protection key located in bits 62:59 of the paging-structure entry that mapped the page containing the linear address; see Section 4.5). Two protection key features control accesses to linear addresses based on their protection keys:

- If CR4.PKE = 1, the PKRU register determines, for each protection key, whether user-mode addresses with that protection key may be read or written.
- If CR4.PKS = 1, the IA32_PKRS MSR (MSR index 6E1H) determines, for each protection key, whether supervisor-mode addresses with that protection key may be read or written.

32-bit paging and PAE paging do not associate linear addresses with protection keys. For the purposes of Section 4.6.1, reads and writes are implicitly permitted for all protection keys with either of those paging modes.

The PKRU register (protection-key rights for user pages) is a 32-bit register with the following format: for each i ($0 \leq i \leq 15$), 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*). The IA32_PKRS MSR has the same format (bits 63:32 of the MSR are reserved and must be zero).

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.

Software can use the RDMSR and WRMSR instructions to read and write the IA32_PKRS MSR. Writes to the IA32_PKRS MSR using WRMSR are not serializing. The IA32_PKRS MSR is not XSAVE-managed.

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.
- If CR4.PKE = 0, the protection key of a user-mode address does not control data accesses to the address (for the purposes of Section 4.6.1, reads and writes of user-mode addresses are implicitly permitted for all protection keys).
 - If CR4.PKE = 1, 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*.)
- If CR4.PKS = 0, the protection key of a supervisor-mode address does not control data accesses to the address (for the purposes of Section 4.6.1, reads and writes of supervisor-mode addresses are implicitly permitted for all protection keys).
 - If CR4.PKS = 1, use of the protection key *i* of a supervisor-mode address depends on the value of the IA32_PKRS MSR:
 - If AD_{*i*} = 1, no data accesses are permitted.
 - If WD_{*i*} = 1, write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, IA32_PKRS.WD_{*i*} does not affect write accesses to supervisor-mode addresses with protection key *i*.)

Protection keys apply to shadow-stack accesses just as they do to ordinary data accesses.

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 37.20, “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:

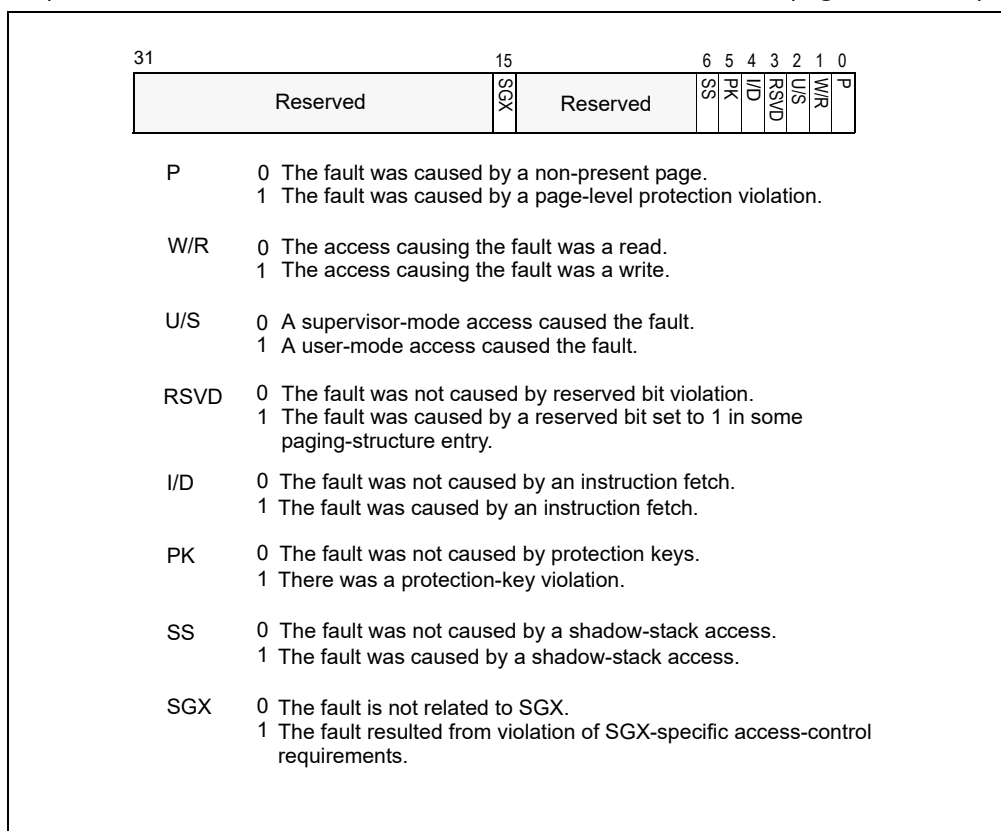


Figure 4-12. Page-Fault Error Code

- **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.

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).

- 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, 4-level paging, or 5-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 only for data accesses and only with 4-level paging and 5-level paging. In these cases, the setting depends on the mode of the address being accessed:
 - For accesses to supervisor-mode addresses, the flag is set if (1) CR4.PKS = 1; (2) the linear address has protection key i ; and (3) the IA32_PKRS MSR (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. (Note that this flag may be set on page faults due to user-mode accesses to supervisor-mode addresses.)
 - For accesses to user-mode addresses, the flag is set if (1) CR4.PKE = 1; (2) the linear address has protection key i ; and (3) 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.
- SS (bit 1).
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), 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.
- 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 PDPTTE 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.¹ 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).

NOTE

If software on one logical processor writes to a page while software on another logical processor concurrently clears the R/W flag in the paging-structure entry that maps the page, execution on some processors may result in the entry's dirty flag being set (due to the write on the first logical processor) and the entry's R/W flag being clear (due to the update to the entry on the second logical processor). This will never occur on a processor that supports control-flow enforcement technology (CET). Specifically, a processor that supports CET will never set the dirty flag in a paging-structure entry in which the R/W flag is clear.

1. With PAE paging, the PDPTTEs are not used during linear-address translation but only to load the PDPTTE registers for some executions of the MOV CR instruction (see Section 4.4.1). For this reason, the PDPTTEs do not contain accessed flags with PAE paging.

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 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 or 5-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 PDPTTE 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 PDPTTEs (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.

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.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry i comprises bits $8i+7:8i$ of the MSR).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 11-11 in Section 11.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry i of the PAT, where i is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with 4-level paging):
 - For 4-level paging or 5-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 PDPTTE 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 PDPTTE (if the translation uses a 1-GByte page).
- With PAE paging, the WB memory type is used when loading the PDPTTEs (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 PDPTTEs. Some processors may use the UC memory type if $CRO.CD = 1$ 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 use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- 4-level paging and 5-level paging:

- If the translation does not use a PDE (because the PS flag is 1 in the PDPTTE 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 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 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 (only with 4-level paging and 5-level paging).
- 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:

- **PML5E cache** (5-level paging only). Each PML5E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 56:40 have that value. The entry contains information from the PML5E used to translate such linear addresses:
 - The physical address from the PML5E (the address of the PML4 table).
 - The value of the R/W flag of the PML5E.
 - The value of the U/S flag of the PML5E.
 - The value of the XD flag of the PML5E.
 - The values of the PCD and PWT flags of the PML5E.

The following items detail how a processor may use the PML5E cache:

 - If the processor has a PML5E-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E in memory).
 - The processor does not create a PML5E-cache entry unless the P flag is 1 and all reserved bits are 0 in the PML5E in memory.
 - The processor does not create a PML5E-cache entry unless the accessed flag is 1 in the PML5E in memory; before caching a translation, the processor sets the accessed flag if it is not already 1.
 - The processor may create a PML5E-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 PML4 table).
 - If the processor creates a PML5E-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E in memory.
- **PML4E cache** (4-level paging and 5-level paging only). The use of the PML4E cache depends on the paging mode:

- For 4-level paging, each PML4E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 47:39 have that value.
- For 5-level paging, each PML4E-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 56:39 have that value.

A PML4E-cache entry contains information from the PML5E and PML4E used to translate the relevant linear addresses (for 4-level paging, the PML5E does not apply):

- The physical address from the PML4E (the address of the page-directory-pointer table).
- The logical-AND of the R/W flags in the PML5E and the PML4E.
- The logical-AND of the U/S flags in the PML5E and the PML4E.
- The logical-OR of the XD flags in the PML5E and the PML4E.
- The values of the PCD and PWT flags of the PML4E.

The following items detail how a processor may use the PML4E cache:

- If the processor has a PML4E-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E and PML4E in memory).
- The processor does not create a PML4E-cache entry unless the P flags are 1 and all reserved bits are 0 in the PML5E and the PML4E in memory.
- The processor does not create a PML4E-cache entry unless the accessed flags are 1 in the PML5E and the PML4E in memory; before caching a translation, the processor sets any accessed flags that are not already 1.
- The processor may create a PML4E-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 PML4E-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E in memory.

- **PDPTÉ cache** (4-level paging and 5-level paging only).¹ The use of the PML4E cache depends on the paging mode:

- For 4-level paging, each PDPTÉ-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 47:30 have that value.
- For 5-level paging, each PDPTÉ-cache entry is referenced by a 27-bit value and is used for linear addresses for which bits 56:30 have that value.

A PDPTÉ-cache entry contains information from the PML5E, PML4E, PDPTÉ used to translate the relevant linear addresses (for 4-level paging, the PML5E does not apply):

- The physical address from the PDPTÉ (the address of the page directory). (No PDPTÉ-cache entry is created for a PDPTÉ that maps a 1-GBYTE page.)
- The logical-AND of the R/W flags in the PML5E, PML4E, and PDPTÉ.
- The logical-AND of the U/S flags in the PML5E, PML4E, and PDPTÉ.
- The logical-OR of the XD flags in the PML5E, PML4E, and PDPTÉ.
- The values of the PCD and PWT flags of the PDPTÉ.

The following items detail how a processor may use the PDPTÉ cache:

- If the processor has a PDPTÉ-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E, PML4E, and PDPTÉ in memory).
- The processor does not create a PDPTÉ-cache entry unless the P flags are 1, the PS flags are 0, and the reserved bits are 0 in the PML5E, PML4E, and PDPTÉ in memory.

1. With PAE paging, the PDPTÉs 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 processor does not create a PDPTÉ-cache entry unless the accessed flags are 1 in the PML5E, PML4E and PDPTÉ in memory; before caching a translation, the processor sets any accessed flags that are not already 1.
- The processor may create a PDPTÉ-cache entry even if there are no translations for any linear address that might use that entry.
- If the processor creates a PDPTÉ-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E, PML4E, or PDPTÉ 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.
 - For 5-level paging, each PDE-cache entry is referenced by a 36-bit value and is used for linear addresses for which bits 56:21 have that value.

A PDE-cache entry contains information from the PML5E, PML4E, PDPTÉ, and PDE used to translate the relevant linear addresses (for 32-bit paging and PAE paging, only the PDE applies; for 4-level paging, the PML5E does not apply):

- 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 PML5E, PML4E, PDPTÉ, and PDE.
- The logical-AND of the U/S flags in the PML5E, PML4E, PDPTÉ, and PDE.
- The logical-OR of the XD flags in the PML5E, PML4E, PDPTÉ, and PDE.
- The values of the PCD and PWT flags of the PDE.

The following items detail how a processor may use the PDE cache (references below to PML5Es, PML4Es, and PDPTÉs apply only to 4-level paging and to 5-level paging, as appropriate):

- 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 PML5E, PML4E, PDPTÉ, and PDE in memory).
- The processor does not create a PDE-cache entry unless the P flags are 1, the PS flags are 0, and the reserved bits are 0 in the PML5E, PML4E, PDPTÉ, and PDE in memory.
- The processor does not create a PDE-cache entry unless the accessed flag is 1 in the PML5E, PML4E, PDPTÉ, and 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 PML5E, PML4E, PDPTÉ, or 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 PDPTÉ-cache entry for a PDPTÉ from the page-directory-pointer table referenced by that PML4E. This is because the R/W flag of each such PDPTÉ-cache entry is the logical-AND of the R/W flags in the appropriate PML4E and PDPTÉ.

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 and 5-level paging, the PDPTe, PML4E, and PML5E, as appropriate) corresponding to the PDE-cache entry.
- The following items apply when 4-level paging or 5-level paging is used:
 - If the processor does not find a relevant TLB entry or PDE-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:30; for 5-level paging, bits 56:30) 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, the PML4E, and (for 5-level paging) the PML5E corresponding to the PDPTe-cache entry.
 - If the processor does not find a relevant TLB entry, PDE-cache entry, or PDPTe-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:39; for 5-level paging, bits 56:39) to select an entry from the PML4E 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.
 - With 5-level paging, if the processor does not find a relevant TLB entry, PDE-cache entry, PDPTe-cache entry, or PML4E-cache entry, it may use bits 56:48 of the linear address to select an entry from the PML5E cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PML4E, etc.) as if it had traversed the corresponding PML5E.

(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 m^{th} PDPTe in the page-directory-pointer table would appear in the PDPTe-cache entries associated with both p_1 and p_2 , where $(p_1 \gg 9) = n_1$, $(p_2 \gg 9) = n_2$, and $(p_1 \& 1\text{FFH}) = (p_2 \& 1\text{FFH}) = m$. 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 $X \gg 12$ 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.
- **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.² (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:

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 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.

- 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;¹ 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.²
- 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.
- 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.³ 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

1. 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.

2. 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.

3. 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.

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.¹

(If the paging-structure entry may be used in the translation of different page numbers — see Section 4.10.3.3 — software should execute INVLPG for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)
- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:
 - Execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLPG at least once.
 - Execute MOV to CR3 if the modified entry controls no global pages.
 - Execute MOV to CR4 to modify CR4.PGE.
- If CR4.PCIDE = 1 and software modifies a paging-structure entry that does not map a page or in which the G flag (bit 8) is 0, additional steps are required if the entry may be used for PCIDs other than the current one. Any one of the following suffices:
 - 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.,

1. One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes.

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 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.)

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.

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.
- 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 shutdown.”

TLB shutdown can be done using memory-based semaphores and/or interprocessor interrupts (IPI). The following items describe a simple but inefficient example of a TLB shutdown algorithm for processors supporting the Intel-64 and IA-32 architectures:

1. If the accesses are to different pages, this may occur even if invalidation has not been delayed.

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 shutdown 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 shutdown 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 or 5-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 sema-

PAGING

phore, 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.

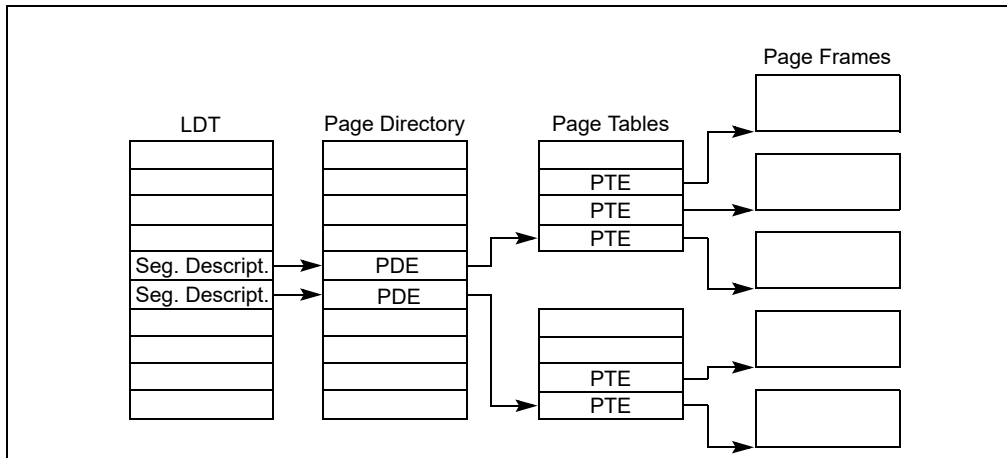


Figure 4-13. Memory Management Convention That Assigns a Page Table to Each Segment

15. Updates to Chapter 5, Volume 3A

Change bars and green text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Added details on 5-level paging details where applicable. Typo corrections as needed.

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.

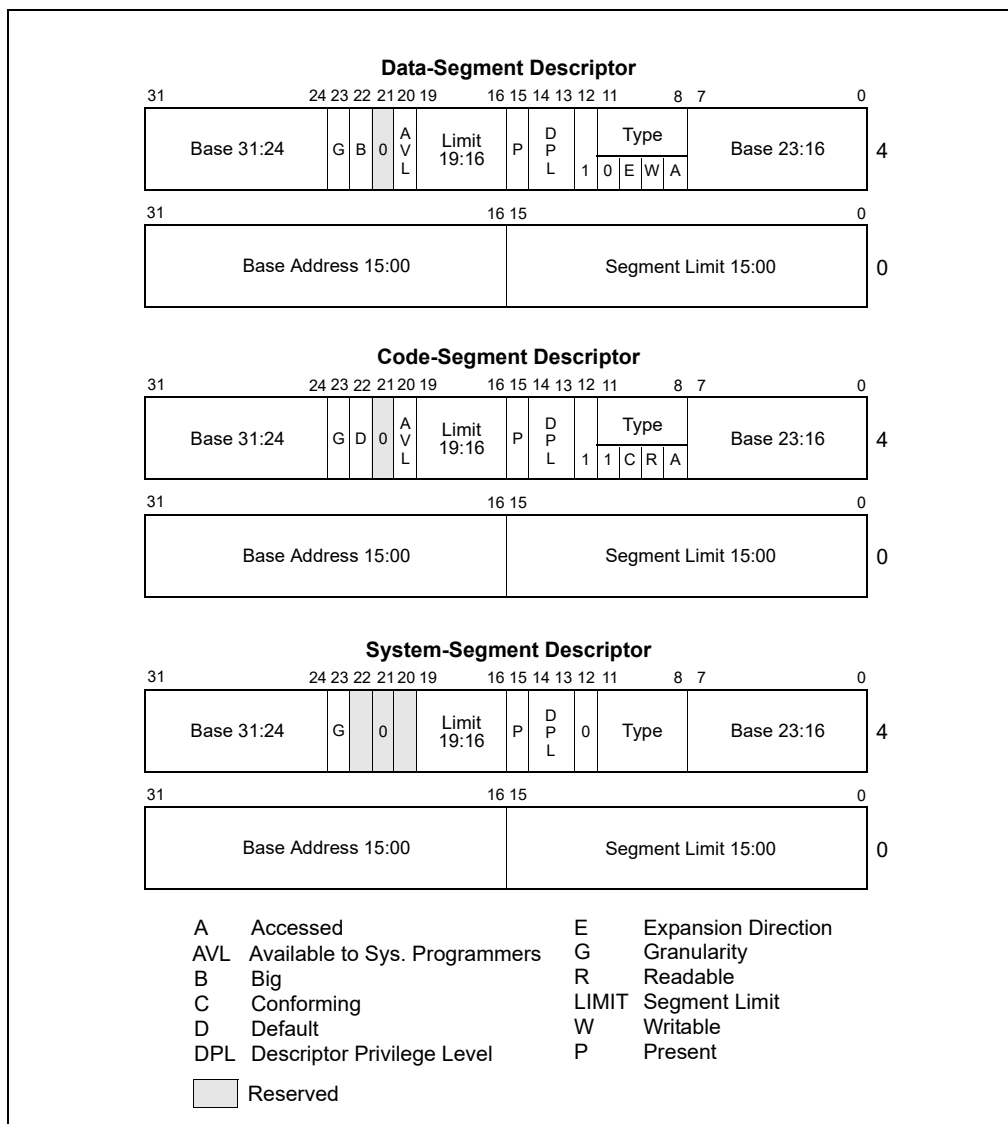


Figure 5-1. Descriptor Fields Used for Protection

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).

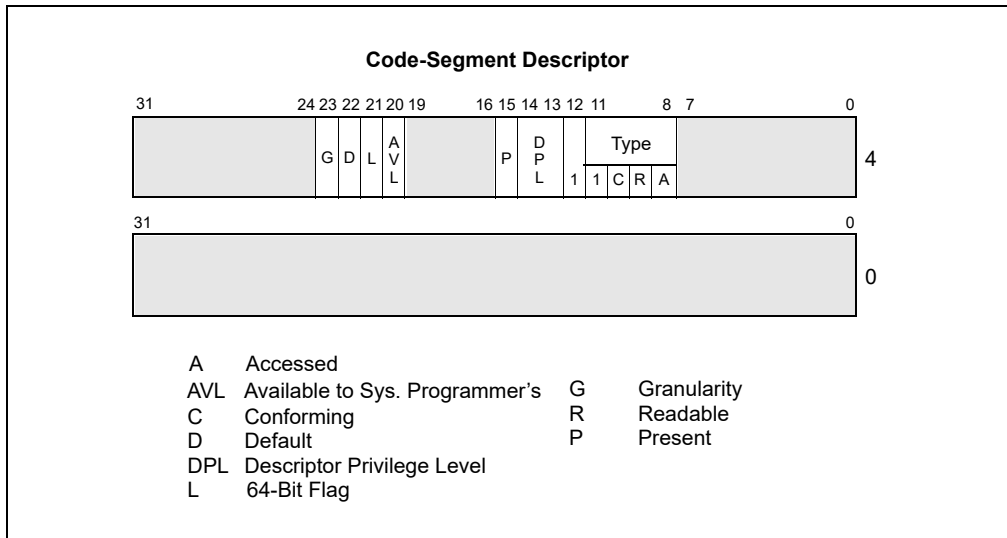


Figure 5-2. Descriptor Fields with Flags used in IA-32e Mode

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.)

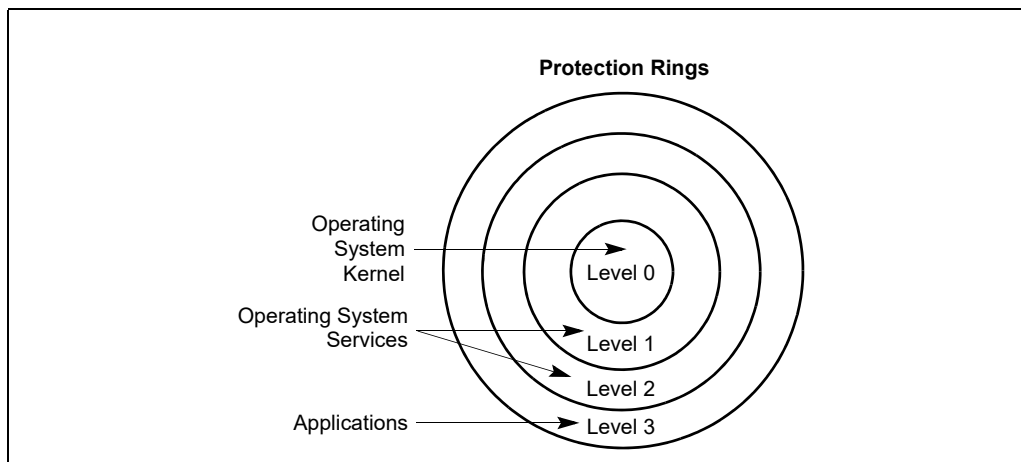


Figure 5-3. Protection Rings

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.

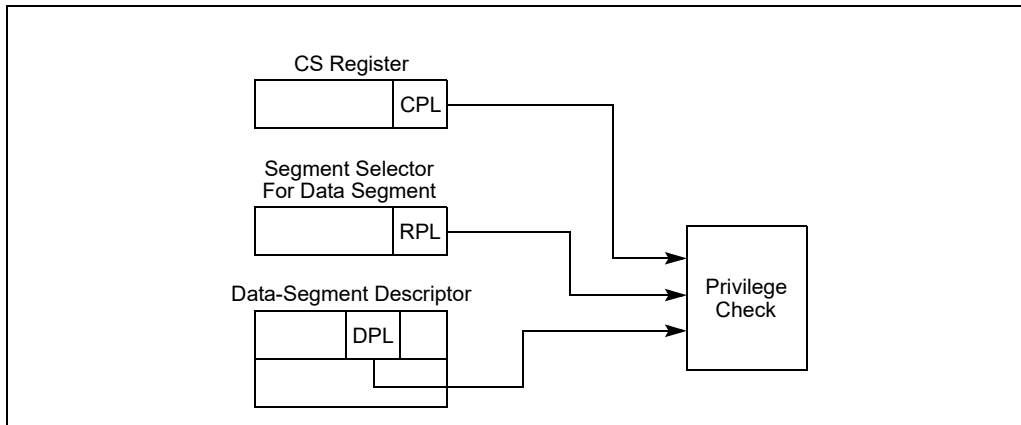


Figure 5-4. Privilege Check for Data Access

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.

- 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.

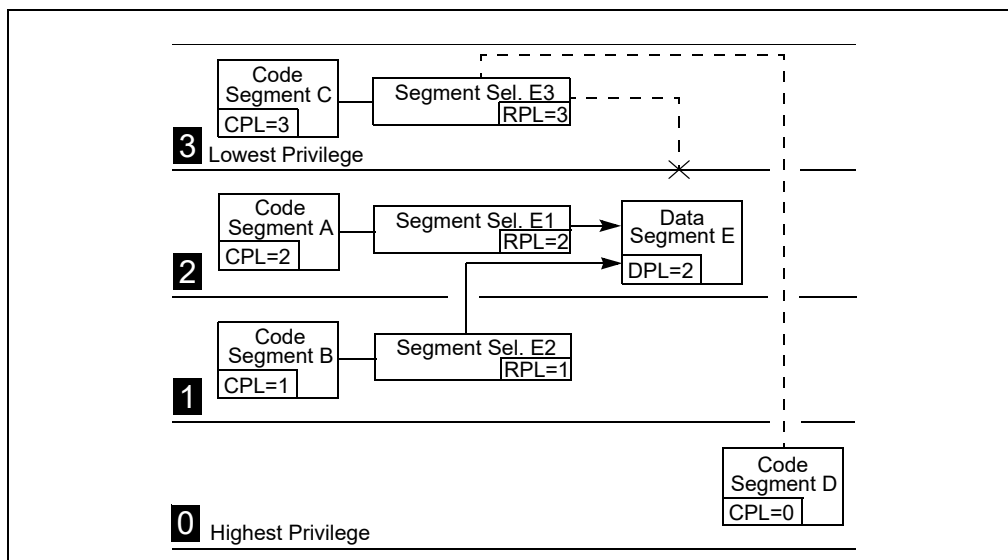


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

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 intentional) 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.)

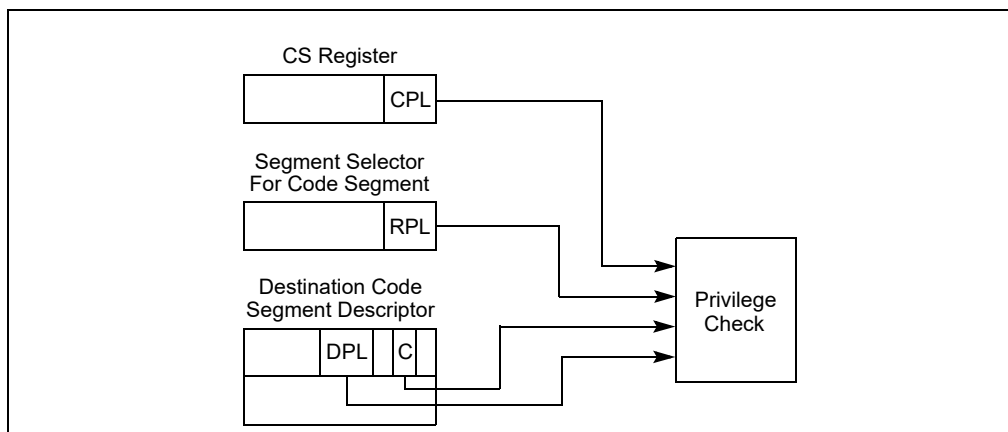


Figure 5-6. Privilege Check for Control Transfer Without Using a Gate

- 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.

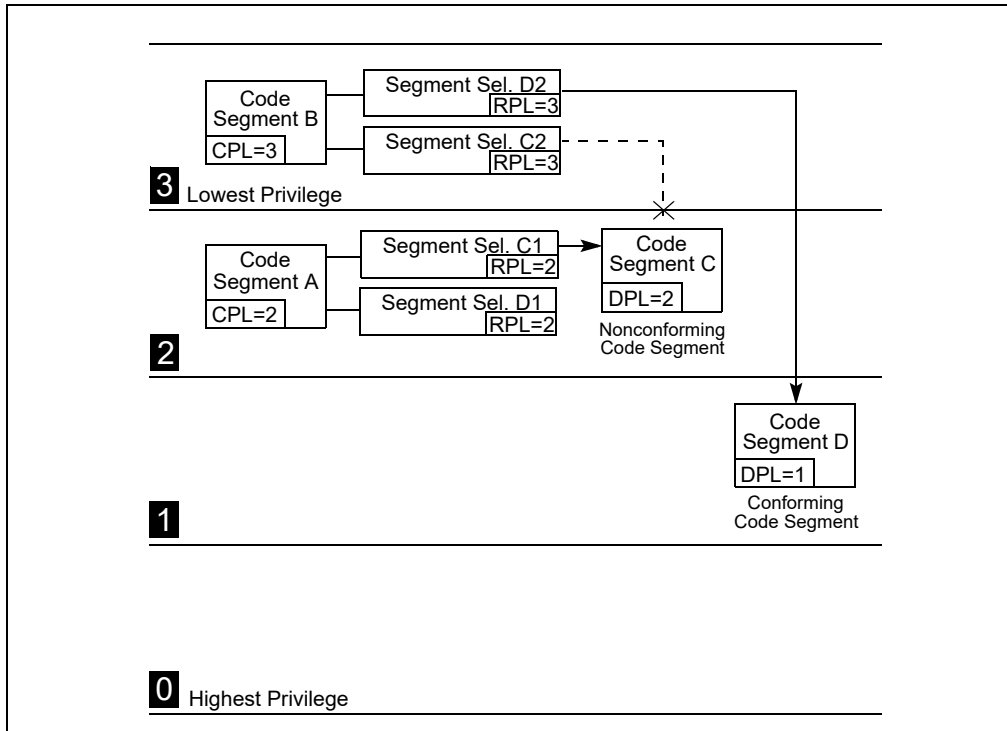


Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

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.

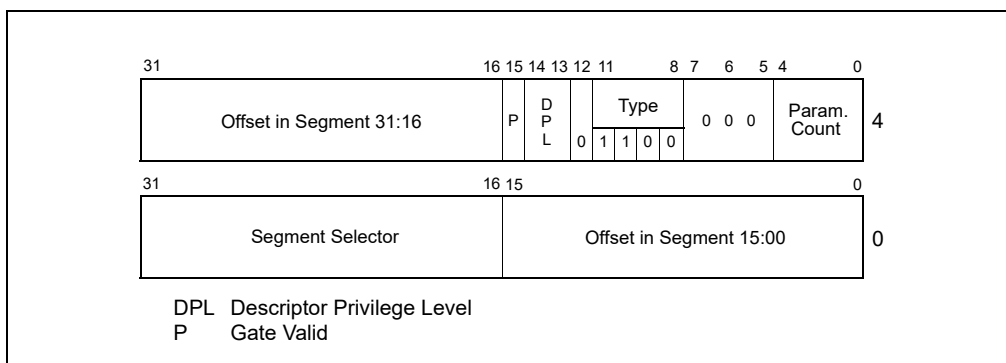


Figure 5-8. Call-Gate Descriptor

- 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.

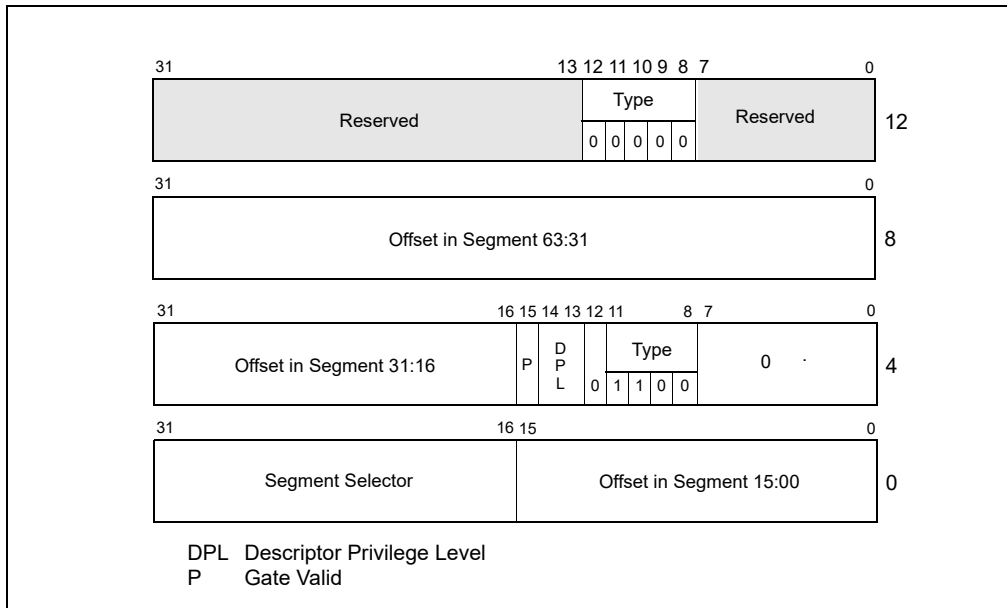


Figure 5-9. Call-Gate Descriptor in IA-32e Mode

- 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.

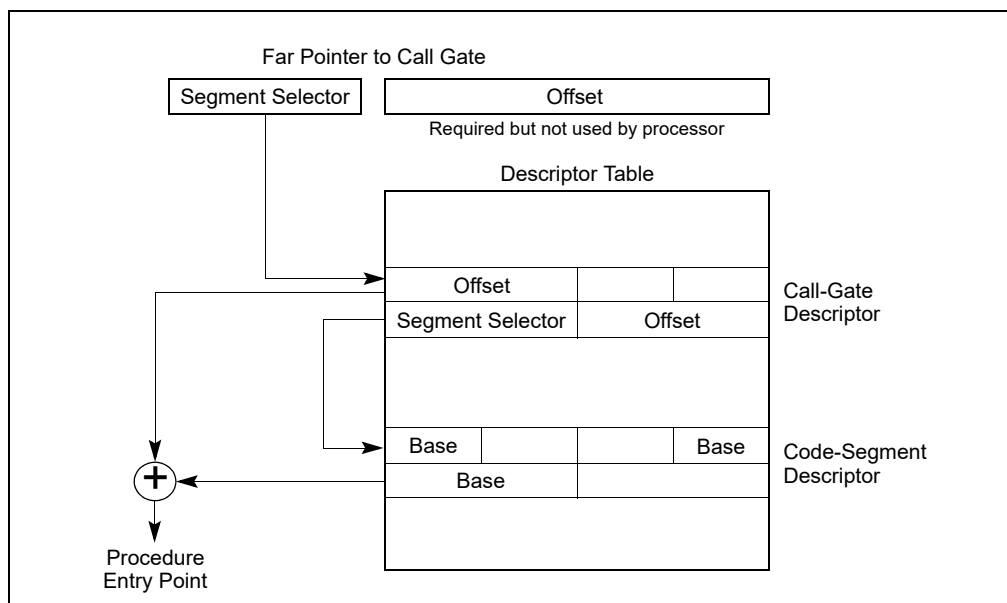


Figure 5-10. Call-Gate Mechanism

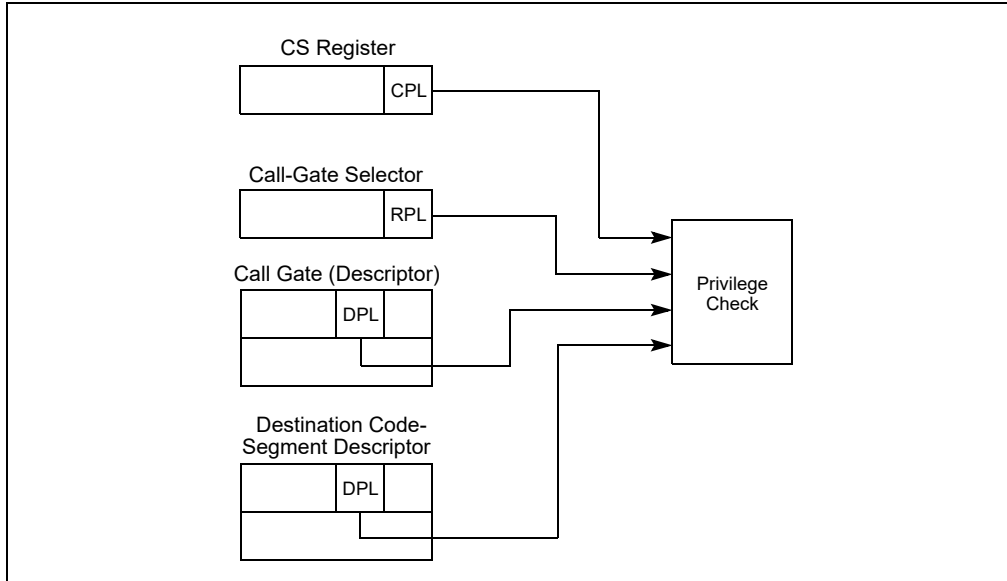


Figure 5-11. Privilege Check for Control Transfer with Call Gate

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	$CPL \leq \text{call gate DPL}; RPL \leq \text{call gate DPL}$ Destination conforming code segment $DPL \leq CPL$ Destination nonconforming code segment $DPL \leq CPL$
JMP	$CPL \leq \text{call gate DPL}; RPL \leq \text{call gate DPL}$ Destination conforming code segment $DPL \leq CPL$ 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 of 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.

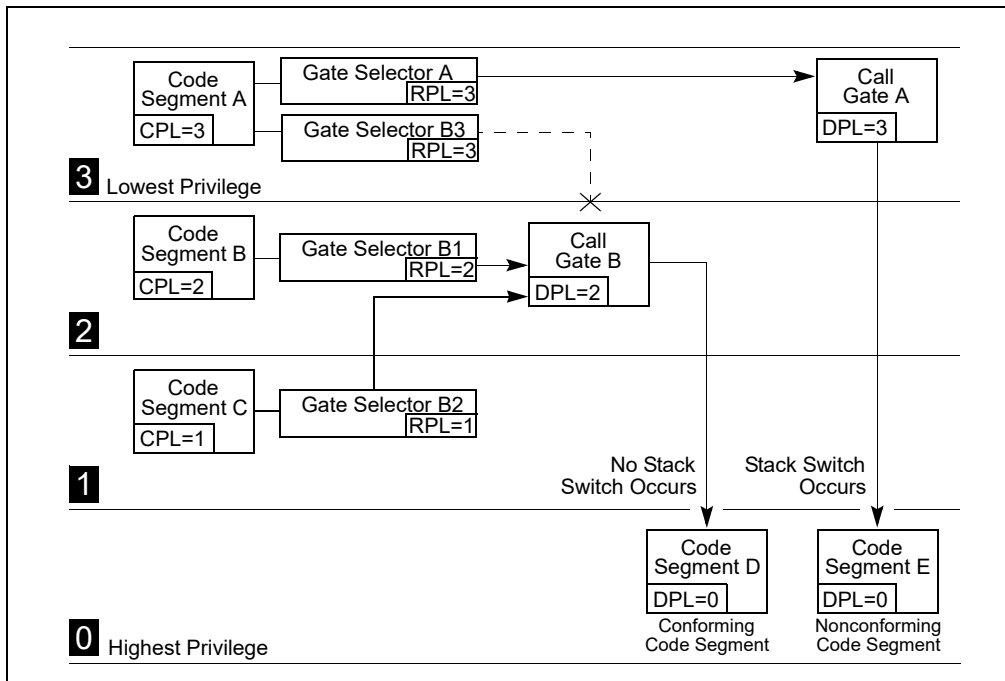


Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels

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.

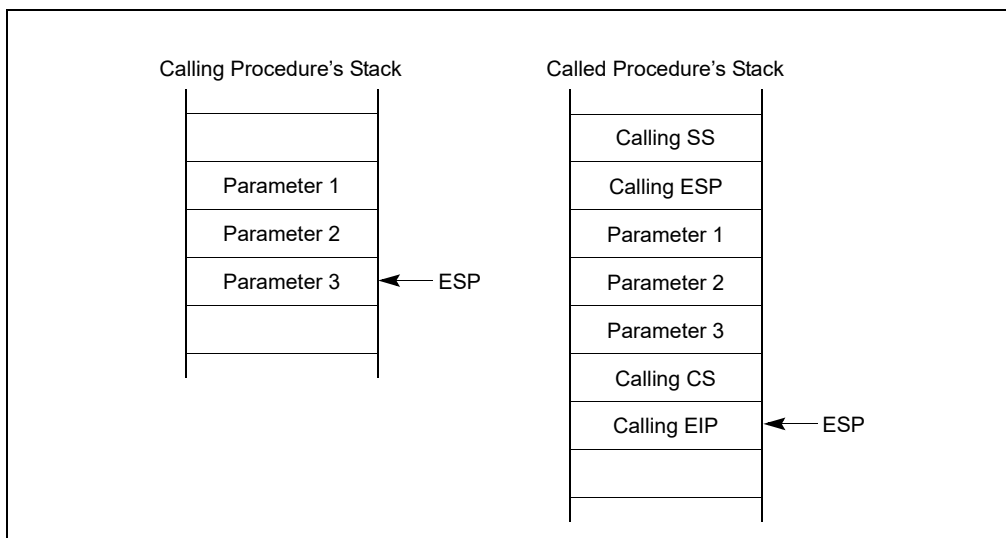


Figure 5-13. Stack Switching During an Interprivilege-Level Call

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		ESP	RSP	IA-32e mode	
Old SS Selector	+12				+24
Old ESP	+8		+16	Old RSP	
CS Selector	+4		+8	Old CS Selector	
EIP	0		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 EDI.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from ECX.

The SYSENTER and SYSEXIT instructions perform “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.

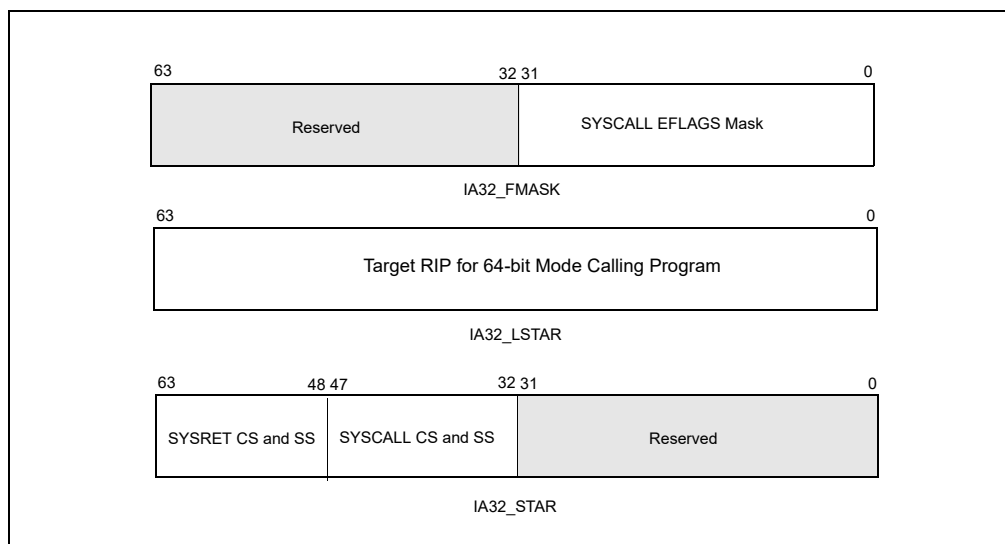


Figure 5-14. MSRs Used by SYSCALL and SYSRET

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 perform 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.

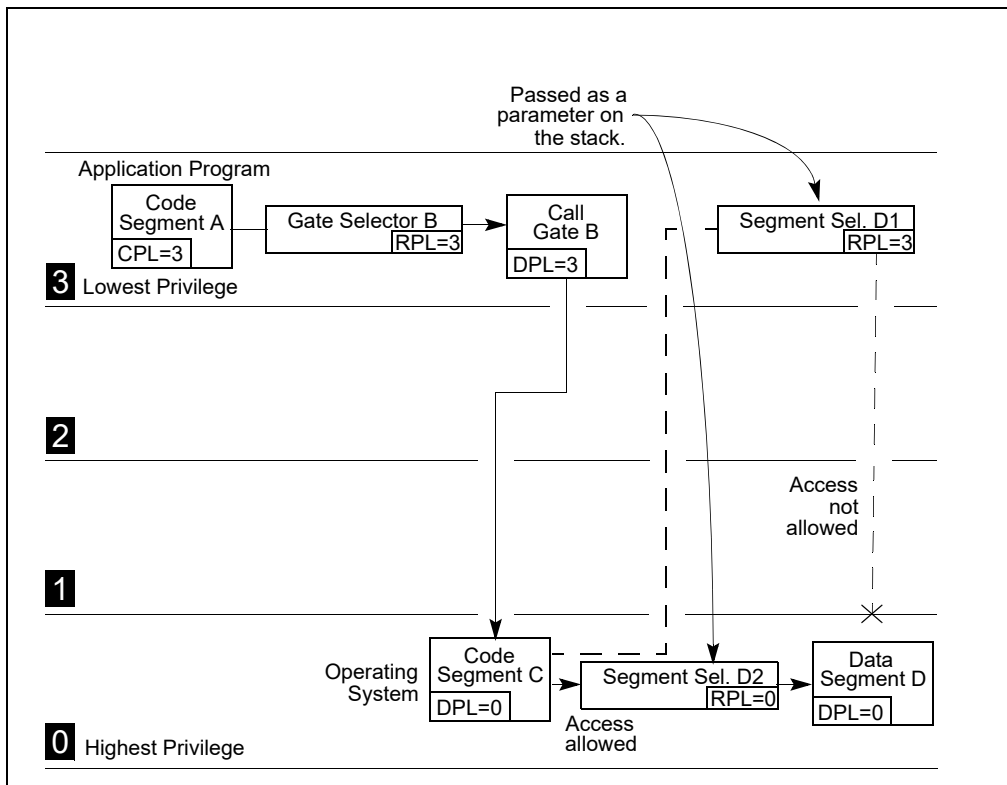


Figure 5-15. Use of RPL to Weaken Privilege Level of Called Procedure

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 an 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 CRO.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. If CRO.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, 4-level paging,¹ and 5-level paging provide the execute-disable bit (see Chapter 4, “Paging”). 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 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. Page Level Protection Matrix with Execute-Disable Bit Capability with 4-Level Paging

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. 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging

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. 2-MByte Page Level Protection with Execute-Disable Bit Capability with PAE Paging

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. 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	PML5E	Bits [51:MAXPHYADDR] *
	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	PML5E	Bit [63], bits [51:MAXPHYADDR]*
	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.

16. Updates to Chapter 6, Volume 3A

Change bars and green text show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Added details on protection keys for supervisor-mode pages where applicable. Typo corrections as needed.

CHAPTER 6

INTERRUPT AND EXCEPTION HANDLING

This chapter describes the interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

Chapter 20, “8086 Emulation,” describes information specific to interrupt and exception mechanisms in real-address and virtual-8086 mode. Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” describes information specific to interrupt and exception mechanisms in IA-32e mode and 64-bit sub-mode.

6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the `INT n` instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor’s interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

6.2 EXCEPTION AND INTERRUPT VECTORS

To aid in handling exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned a unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT). The table provides the entry point to an exception or interrupt handler (see Section 6.10, “Interrupt Descriptor Table (IDT)”).

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture-defined exceptions and interrupts. Not all of the vector numbers in this range have a currently defined function. The unassigned vector numbers in this range are reserved. Do not use the reserved vector numbers.

Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are not reserved by the Intel 64 and IA-32 architecture. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms (see Section 6.3, “Sources of Interrupts”).

Table 6-1 shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (see Section 6.5, "Exception Classifications") and indicates whether an error code is saved on the stack for the exception. The source of each predefined exception and the NMI interrupt is also given.

6.3 SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

6.3.1 External Interrupts

External interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the local APIC (see Chapter 10, "Advanced Programmable Interrupt Controller (APIC)"). When the local APIC is enabled, the LINT[1:0] pins can be programmed through the APIC's local vector table (LVT) to be associated with any of the processor's exception or interrupt vectors.

When the local APIC is global/hardware disabled, these pins are configured as INTR and NMI pins, respectively. Asserting the INTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A (see Section 6.2, "Exception and Interrupt Vectors"). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. ¹
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

Vector	Mnemonic	Description	Type	Error Code	Source
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory. ²
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent. ³
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions ⁴
20	#VE	Virtualization Exception	Fault	No	EPT violations ⁵
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT <i>n</i> instruction.

NOTES:

- Processors after the Intel386 processor do not generate this exception.
- This exception was introduced in the Intel486 processor.
- This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
- This exception was introduced in the Pentium III processor.
- This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.

The processor’s local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC’s pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom®, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 34, “System Management Mode.”

6.3.2 Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include

all IA-32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local APIC include interrupt vectors 16 through 255.

The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group (see Section 6.8.1, “Masking Maskable Hardware Interrupts”). Note that when interrupts 0 through 15 are delivered through the local APIC, the APIC indicates the receipt of an illegal vector.

6.3.3 Software-Generated Interrupts

The INT *n* instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand. For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. If the processor’s predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. If vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor’s NMI-handling hardware is not activated.

Interrupts generated in software with the INT *n* instruction cannot be masked by the IF flag in the EFLAGS register.

6.4 SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

6.4.1 Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor-detectable exception. Exceptions are classified as **faults**, **traps**, and **aborts** (see Section 6.5, “Exception Classifications”).

6.4.2 Software-Generated Exceptions

The INTO, INT1, INT3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT3 causes a breakpoint exception to be generated.

The INT *n* instruction can be used to emulate exceptions in software; but there is a limitation.¹ If INT *n* provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

6.4.3 Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation depen-

1. The INT *n* instruction has opcode CD following by an immediate byte encoding the value of *n*. In contrast, INT1 has opcode F1 and INT3 has opcode CC.

dent. When a machine-check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code.

See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC)” and Chapter 15, “Machine-Check Architecture,” for more information about the machine-check mechanism.

6.5 EXCEPTION CLASSIFICATIONS

Exceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

NOTE

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. In this situation, the exception handler sees that the instruction pointer (CS:EIP) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

6.6 PROGRAM OR TASK RESTART

To allow the restarting of program or task following the handling of an exception or an interrupt, all exceptions (except aborts) are guaranteed to report exceptions on an instruction boundary. All interrupts are guaranteed to be taken on an instruction boundary.

For fault-class exceptions, the return instruction pointer (saved when the processor generates an exception) points to the faulting instruction. So, when a program or task is restarted following the handling of a fault, the faulting instruction is restarted (re-executed). Restarting the faulting instruction is commonly used to handle exceptions that are generated when access to an operand is blocked. The most common example of this type of fault is a page-fault exception (#PF) that occurs when a program or task references an operand located on a page that is not in memory. When a page-fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To insure that the restart is handled transparently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

For trap-class exceptions, the return instruction pointer points to the instruction following the trapping instruction. If a trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction. All trap exceptions allow program or task restart with no loss of continuity. For example, the overflow exception is a trap exception. Here, the return instruction pointer points to the instruction following the INTO instruction that tested EFLAGS.OF (overflow) flag. The trap handler for this exception resolves the overflow condition. Upon return from the trap handler, program or task execution continues at the instruction following the INTO instruction.

The abort-class exceptions do not support reliable restarting of the program or task. Abort handlers are designed to collect diagnostic information about the state of the processor when the abort exception occurred and then shut down the application and system as gracefully as possible.

Interrupts rigorously support restarting of interrupted programs and tasks without loss of continuity. The return instruction pointer saved for an interrupt points to the next instruction to be executed at the instruction boundary where the processor took the interrupt. If the instruction just executed has a repeat prefix, the interrupt is taken at the end of the current iteration with the registers set to execute the next iteration.

The ability of a P6 family processor to speculatively execute instructions does not affect the taking of interrupts by the processor. Interrupts are taken at instruction boundaries located during the retirement phase of instruction execution; so they are always taken in the “in-order” instruction stream. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the P6 family processors’ microarchitecture and its support for out-of-order instruction execution.

Note that the Pentium processor and earlier IA-32 processors also perform varying amounts of prefetching and preliminary decoding. With these processors as well, exceptions and interrupts are not signaled until actual “in-order” execution of the instructions. For a given code sample, the signaling of exceptions occurs uniformly when the code is executed on any family of IA-32 processors (except where new exceptions or new opcodes have been defined).

6.7 NONMASKABLE INTERRUPT (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors) with a delivery mode NMI.

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to insure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, “Handling Multiple NMIs”).

Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register.

It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor’s NMI-handling hardware can only be delivered through one of the mechanisms listed above.

6.7.1 Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMIs until the next execution of the IRET instruction. This blocking of NMIs prevents nested execution of the NMI handler. It is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts (see Section 6.8.1, “Masking Maskable Hardware Interrupts”).

An execution of the IRET instruction unblocks NMIs even if the instruction causes a fault. For example, if the IRET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated (see Section 20.2.7, “Sensitive Instructions”). In such a case, NMIs are unmasked before the exception handler is invoked.

6.8 ENABLING AND DISABLING INTERRUPTS

The processor inhibits the generation of some interrupts, depending on the state of the processor and of the IF and RF flags in the EFLAGS register, as described in the following sections.

6.8.1 Masking Maskable Hardware Interrupts

The IF flag can disable the servicing of maskable hardware interrupts received on the processor's INTR pin or through the local APIC (see Section 6.3.2, "Maskable Hardware Interrupts"). When the IF flag is clear, the processor inhibits interrupts delivered to the INTR pin or through the local APIC from generating an internal interrupt request; when the IF flag is set, interrupts delivered to the INTR or through the local APIC pin are processed as normal external interrupts.

The IF flag does not affect non-maskable interrupts (NMIs) delivered to the NMI pin or delivery mode NMI messages delivered through the local APIC, nor does it affect processor generated exceptions. As with the other flags in the EFLAGS register, the processor clears the IF flag in response to a hardware reset.

The fact that the group of maskable hardware interrupts includes the reserved interrupt and exception vectors 0 through 32 can potentially cause confusion. Architecturally, when the IF flag is set, an interrupt for any of the vectors from 0 through 32 can be delivered to the processor through the INTR pin and any of the vectors from 16 through 32 can be delivered through the local APIC. The processor will then generate an interrupt and call the interrupt or exception handler pointed to by the vector number. So for example, it is possible to invoke the page-fault handler through the INTR pin (by means of vector 14); however, this is not a true page-fault exception. It is an interrupt. As with the INT *n* instruction (see Section 6.4.2, "Software-Generated Exceptions"), when an interrupt is generated through the INTR pin to an exception vector, the processor does not push an error code on the stack, so the exception handler may not operate correctly.

The IF flag can be set or cleared with the STI (set interrupt-enable flag) and CLI (clear interrupt-enable flag) instructions, respectively. These instructions may be executed only if the CPL is equal to or less than the IOPL. A general-protection exception (#GP) is generated if they are executed when the CPL is greater than the IOPL.² If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI.³ The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag is also affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified flags back into the EFLAGS register.
- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is handled through an interrupt gate, the IF flag is automatically cleared, which disables maskable hardware interrupts. (If an interrupt is handled through a trap gate, the IF flag is not cleared.)

See the descriptions of the CLI, STI, PUSHF, POPF, and IRET instructions in Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and Chapter 4, "Instruction Set Reference, M-U," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a detailed description of the operations these instructions are allowed to perform on the IF flag.

6.8.2 Masking Instruction Breakpoints

The RF (resume) flag in the EFLAGS register controls the response of the processor to instruction-breakpoint conditions (see the description of the RF flag in Section 2.3, "System Flags and Fields in the EFLAGS Register").

When set, it prevents an instruction breakpoint from generating a debug exception (#DB); when clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to prevent the processor from going into a debug exception loop on an instruction-breakpoint. See Section 17.3.1.1, "Instruction-Breakpoint Exception Condition," for more information on the use of this flag.

As noted in Section 6.8.3, execution of the MOV or POP instruction to load the SS register suppresses any instruction breakpoint on the next instruction (just as if EFLAGS.RF were 1).

2. The effect of the IOPL on these instructions is modified slightly when the virtual mode extension is enabled by setting the VME flag in control register CR4: see Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode." Behavior is also impacted by the PVI flag: see Section 20.4, "Protected-Mode Virtual Interrupts."

3. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.

6.8.3 Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, StackTop
```

(Software might also use the POP instruction to load SS and ESP.)

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

To account for this situation, the processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction. The following items provide details:

- Any instruction breakpoint on the next instruction is suppressed (as if EFLAGS.RF were 1).
- Any data breakpoint on the MOV to SS instruction or POP to SS instruction is inhibited until the instruction boundary following the next instruction.
- Any single-step trap that would be delivered following the MOV to SS instruction or POP to SS instruction (because EFLAGS.TF is 1) is suppressed.
- The suppression and inhibition ends after delivery of an exception or the execution of the next instruction.
- If a sequence of consecutive instructions each loads the SS register (using MOV or POP), only the first is guaranteed to inhibit or suppress events in this way.

Intel recommends that software use the LSS instruction to load the SS register and ESP together. The problem identified earlier does not apply to LSS, and the LSS instruction does not inhibit events as detailed above.

6.9 PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS

If more than one exception or interrupt is pending at an instruction boundary, the processor services them in a predictable order. Table 6-2 shows the priority among classes of exception and interrupt sources.

Table 6-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) ¹
6	Maskable Hardware Interrupts ¹
7	Code Breakpoint Fault

Table 6-2. Priority Among Simultaneous Exceptions and Interrupts (Contd.)

Priority	Description
8	Faults from Fetching Next Instruction <ul style="list-style-type: none"> - Code-Segment Limit Violation - Code Page Fault - Control protection exception due to missing ENDBRANCH at target of an indirect call or jump
9	Faults from Decoding the Next Instruction <ul style="list-style-type: none"> - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
10 (Lowest)	Faults on Executing an Instruction <ul style="list-style-type: none"> - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception - Virtualization exception - Control protection exception

NOTE

1. The Intel® 486 processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

While priority among these classes listed in Table 6-2 is consistent throughout the architecture, exceptions within each class are implementation-dependent and may vary from processor to processor. The processor first services a pending exception or interrupt from the class which has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions are re-generated when the interrupt handler returns execution to the point in the program or task where the exceptions and/or interrupts occurred.

6.10 INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors (in protected mode). Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the IDT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, $8N - 1$).

The IDT may reside anywhere in the linear address space. As shown in Figure 6-1, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SIDT (store IDT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another. The SIDT instruction copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

If a vector references a descriptor beyond the limit of the IDT, a general-protection exception (#GP) is generated.

NOTE

Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery.

IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. The same apply for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.

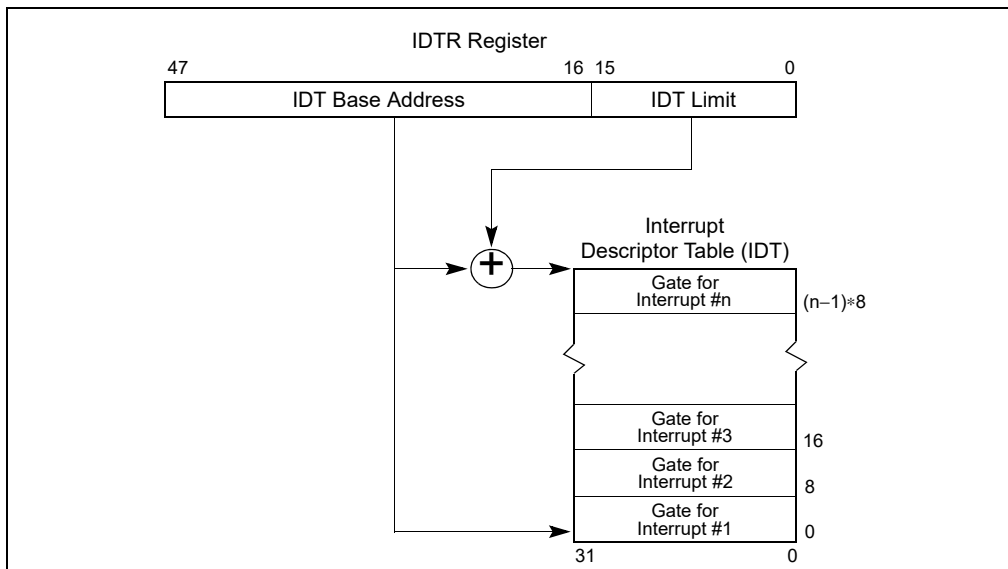


Figure 6-1. Relationship of the IDTR and IDT

6.11 IDT DESCRIPTORS

The IDT may contain any of three kinds of gate descriptors:

- Task-gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Figure 6-2 shows the formats for the task-gate, interrupt-gate, and trap-gate descriptors. The format of a task gate used in an IDT is the same as that of a task gate used in the GDT or an LDT (see Section 7.2.5, "Task-Gate Descriptor"). The task gate contains the segment selector for a TSS for an exception and/or interrupt handler task.

Interrupt and trap gates are very similar to call gates (see Section 5.8.3, "Call Gates"). They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt-handler code segment. These gates differ in the way the processor handles the IF flag in the EFLAGS register (see Section 6.12.1.3, "Flag Usage By Exception- or Interrupt-Handler Procedure").

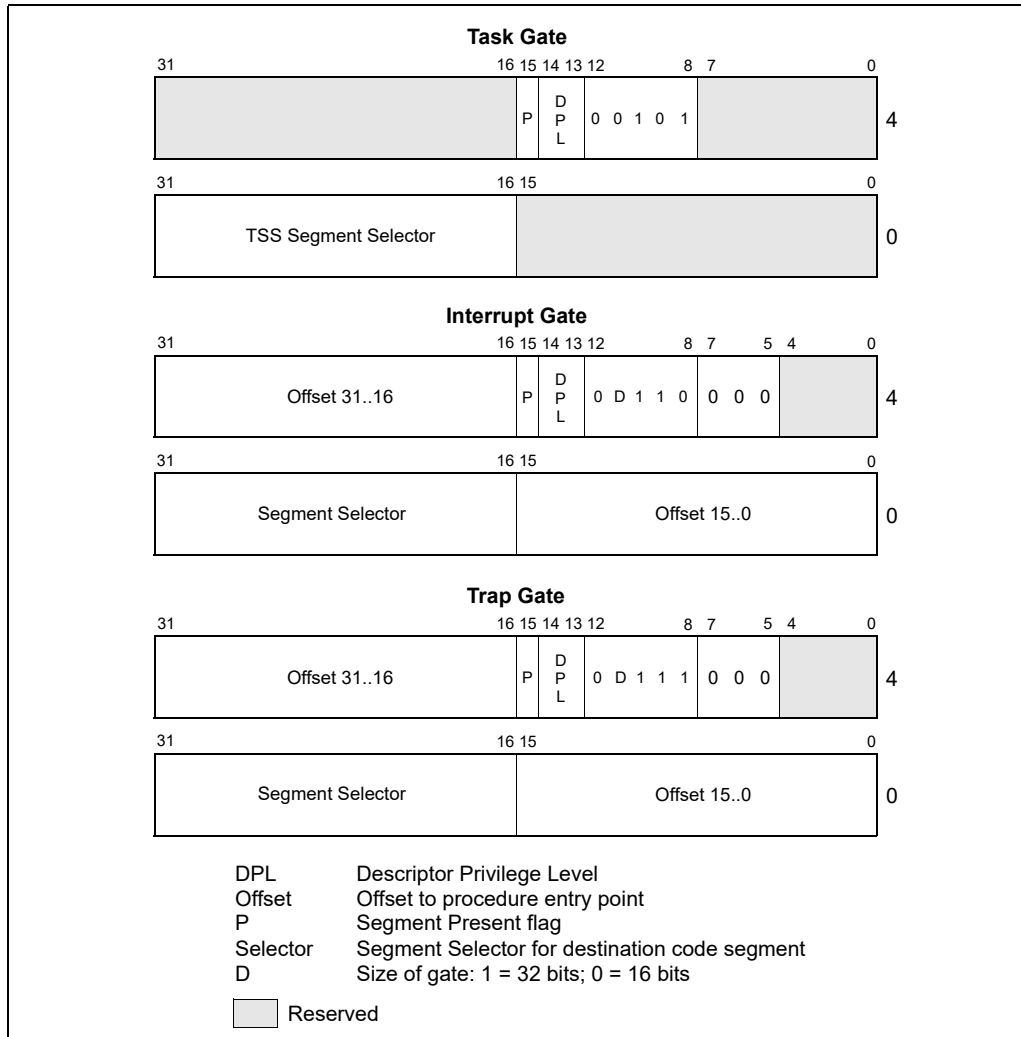


Figure 6-2. IDT Gate Descriptors

6.12 EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 7.3, "Task Switching").

6.12.1 Exception- or Interrupt-Handler Procedures

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task (see Figure 6-3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt-handling procedure.

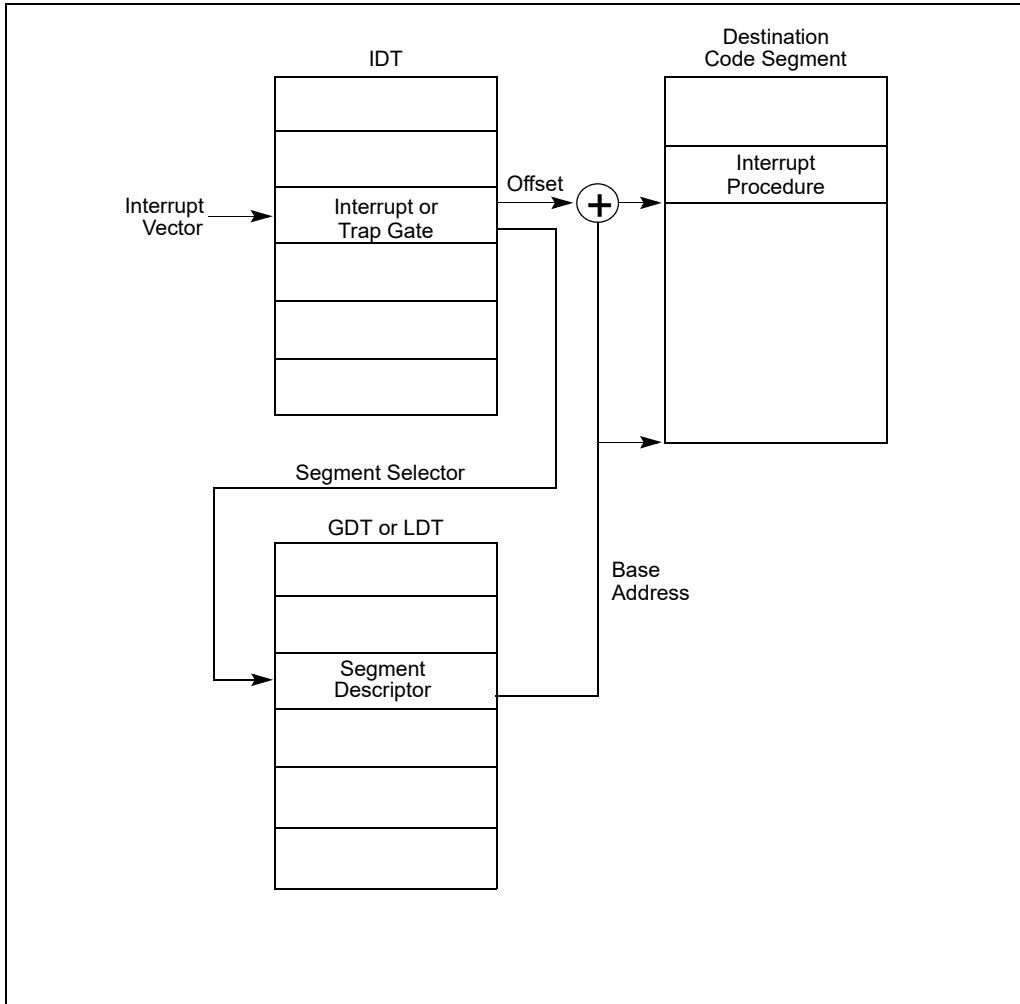


Figure 6-3. Interrupt Procedure Call

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
 - a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
 - b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 6-4).
 - c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
 - a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 6-4).
 - b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

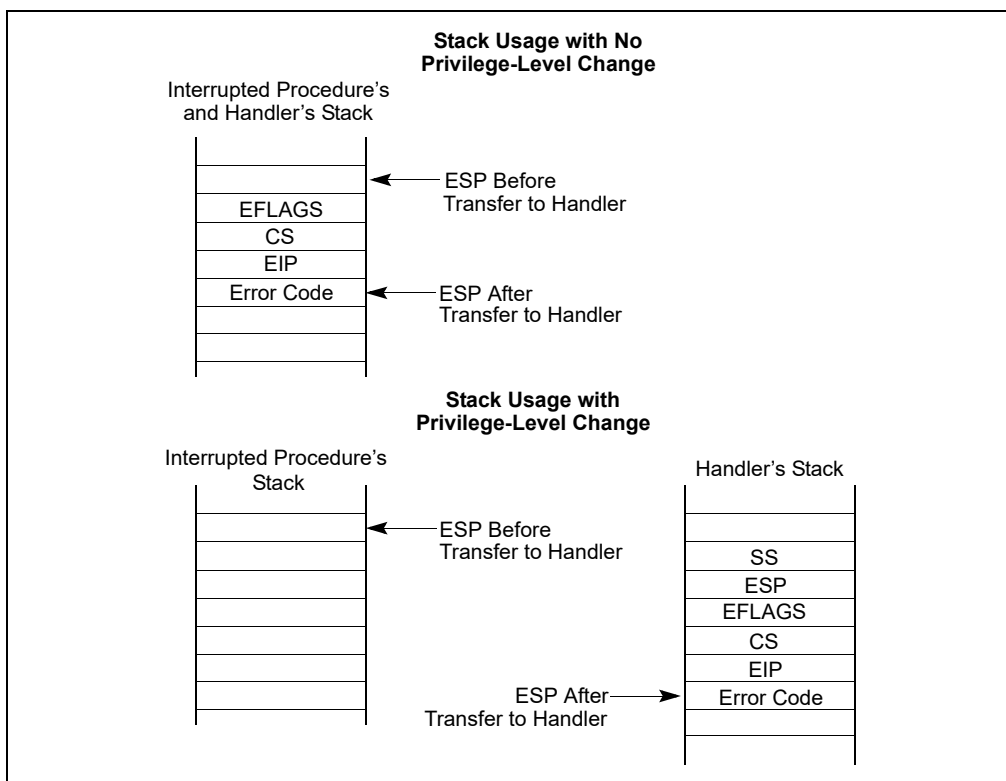


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for a description of the complete operation performed by the IRET instruction.

If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

6.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a shadow stack switch occurs. When the shadow stack switch occurs:
 - a. On a transfer from privilege level 3, if shadow stacks are enabled at privilege level 3 then the SSP is saved to the IA32_PL3_SSP MSR.
 - b. If shadow stacks are enabled at the privilege level where the handler will execute then the shadow stack for the handler is obtained from one of the following MSRs based on the privilege level at which the handler executes.
 - IA32_PL2_SSP if handler executes at privilege level 2.
 - IA32_PL1_SSP if handler executes at privilege level 1.
 - IA32_PL0_SSP if handler executes at privilege level 0.
 - c. The SSP obtained is then verified to ensure it points to a valid supervisory shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 18.2.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
 - d. On this new shadow stack, the processor pushes the CS, LIP (CS.base + EIP), and SSP of the interrupted procedure if the interrupted procedure was executing at privilege level less than 3; see Figure 6-5.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure and shadow stacks are enabled at current privilege level:
 - a. The processor saves the current state of the CS, LIP (CS.base + EIP), and SSP registers on the current shadow stack; see Figure 6-5.

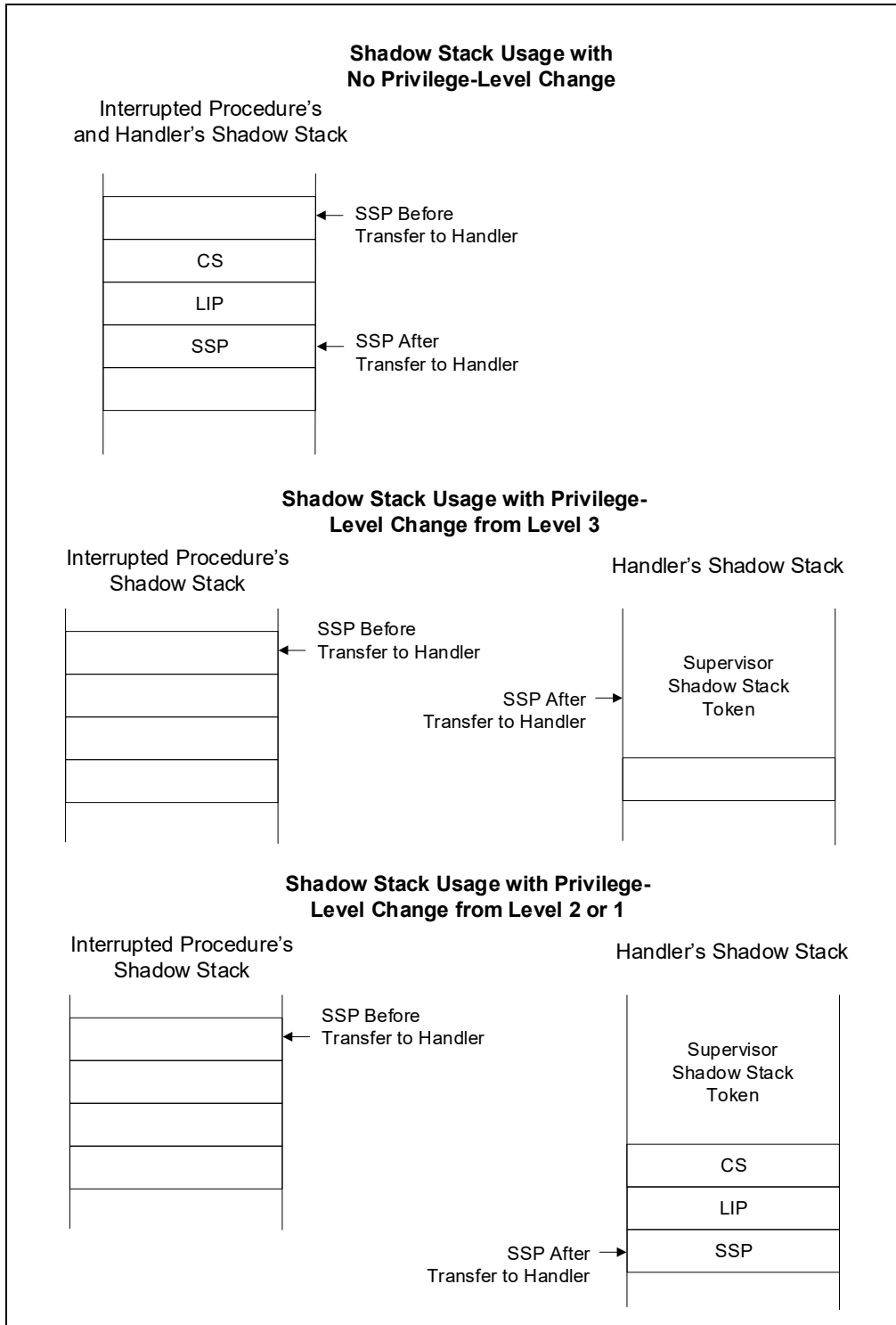


Figure 6-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions to enforce return address protection:

- Restores the CS and EIP registers to their values prior to the interrupt or exception.

If shadow stack is enabled:

- Compares the values on shadow stack at address $SSP+8$ (the LIP) and $SSP+16$ (the CS) to the CS and $(CS.base + EIP)$ popped from the stack and causes a control protection exception ($\#CP(FAR-RET/IRET)$) if they do not match.
- Pops the top-of-stack value (the SSP prior to the interrupt or exception) from shadow stack into SSP register.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs the actions below.

- If shadow stack is enabled at current privilege level:
 - If SSP is not aligned to 8 bytes then causes a control protection exception ($\#CP(FAR-RET/IRET)$).
 - If privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
 - Compares the values on shadow stack at address $SSP+8$ (the LIP) and $SSP+16$ (the CS) to the CS and $(CS.base + EIP)$ popped from the stack and causes a control protection exception ($\#CP(FAR-RET/IRET)$) if they do not match.
 - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
 - If a busy supervisor shadow stack token is present at address $SSP+24$, then marks the token free using operations described in section Section 18.2.3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
 - If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
 - If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with value of IA32_PL3_SSP MSR.

6.12.1.2 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 5.8.4, "Accessing a Code Segment Through a Call Gate"). The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general-protection exception ($\#GP$). The protection mechanism for exception- and interrupt-handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.
- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an $INT\ n$, $INT3$, or $INTO$ instruction.⁴ Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are placed in more privileged code segments (numerically lower privilege level). For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt- handling procedures can run. Either of the following techniques can be used to avoid privilege-level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

4. This check is not performed by execution of the $INT1$ instruction (opcode $F1$); it would be performed by execution of $INT\ 1$ (opcode $CD\ 01$).

6.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response and ensures that no single-step exception will be delivered after delivery to the handler. A subsequent IRET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register. When accessing an exception- or interrupt-handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

6.12.2 Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency.

A task gate in the IDT references a TSS descriptor in the GDT (see Figure 6-6). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 7.3, "Task Switching"). The link back to the interrupted task is stored in the previous task link field of the handler task's TSS. If an exception caused an error code to be generated, this error code is copied to the stack of the new task.

When exception- or interrupt-handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism). The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.

NOTE

Because IA-32 architecture tasks are not re-entrant, an interrupt-handler task must disable interrupts between the time it completes handling the interrupt and the time it executes the IRET instruction. This action prevents another interrupt from occurring while the interrupt task's TSS is still marked busy, which would cause a general-protection (#GP) exception.

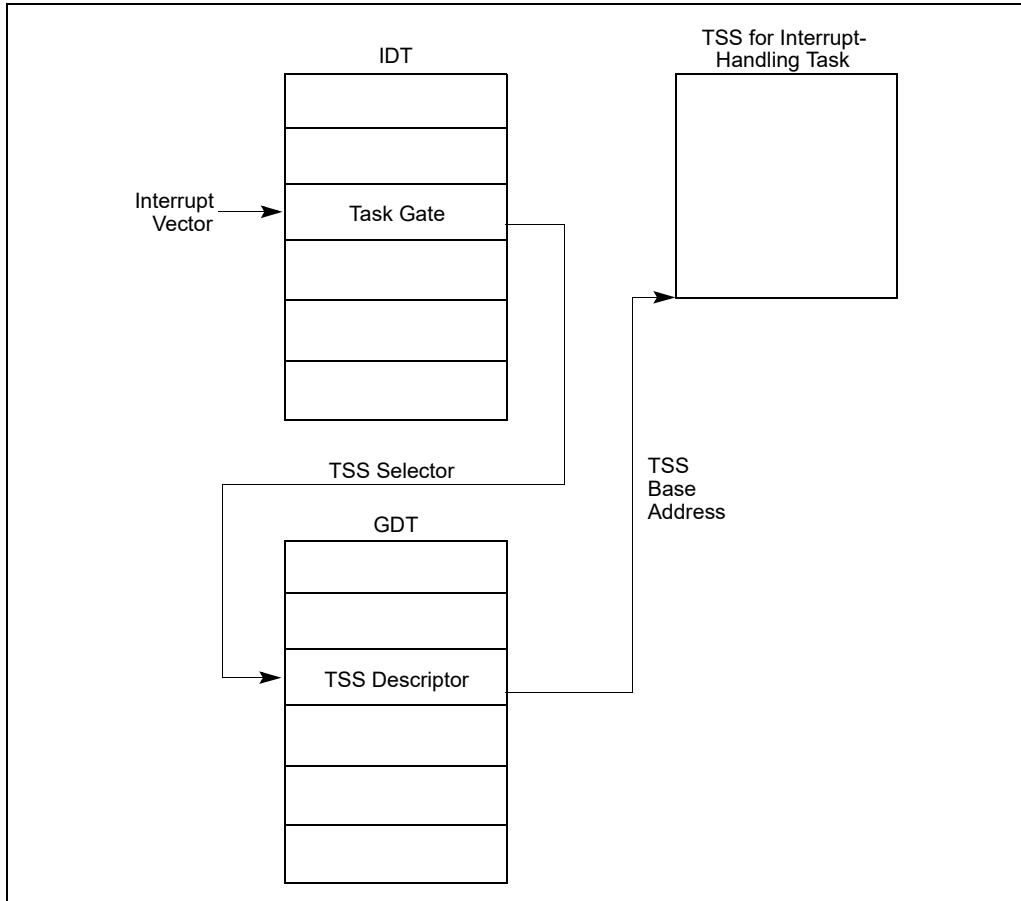


Figure 6-6. Interrupt Task Switch

6.13 ERROR CODE

When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure 6-7. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

- EXT** **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.⁵ The bit is cleared if the exception occurred during delivery of a software interrupt (INT *n*, INT3, or INTO).
- IDT** **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.
- TI** **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

5. The bit is also set if the exception occurred during delivery of INT1.

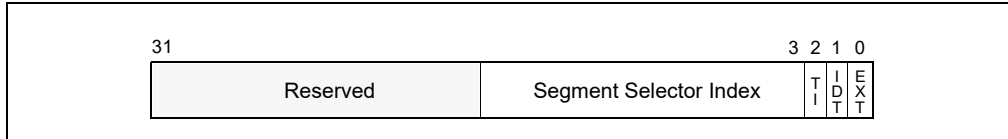


Figure 6-7. Error Code

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment selector was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the “Interrupt 14—Page-Fault Exception (#PF)” section in this chapter.

The format of the error code is different for control protection exceptions (#CP). See the “Interrupt 21—Control Protection Exception (#CP)” section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INT *n* instruction, even if an error code is normally produced for those exceptions.

6.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE

In 64-bit mode, interrupt and exception handling is similar to what has been described for non-64-bit modes. The following are the exceptions:

- All interrupt handlers pointed by the IDT are in 64-bit code (this does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits; and the processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy modes, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism and a new interrupt shadow stack-switch mechanism.
- The alignment of interrupt stack frame is different.

6.14.1 64-Bit Mode IDT

Interrupt and trap gates are 16 bytes in length to provide a 64-bit offset for the instruction pointer (RIP). The 64-bit RIP referenced by interrupt-gate descriptors allows an interrupt service routine to be located anywhere in the linear-address space. See Figure 6-8.

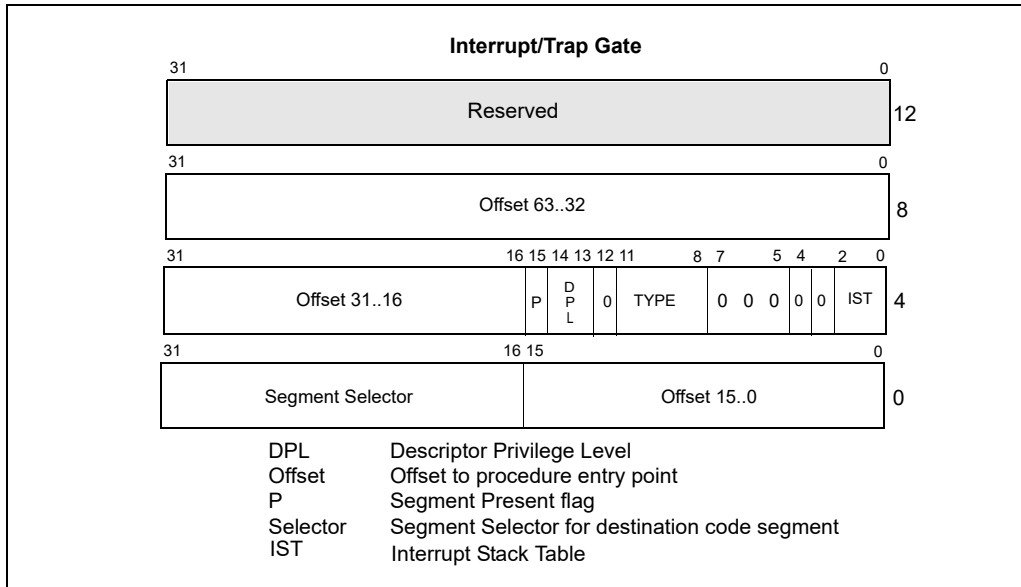


Figure 6-8. 64-Bit IDT Gate Descriptors

In 64-bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7:0) of a 64-bit mode interrupt gate are similar but not identical to legacy 32-bit interrupt gates. The type field (bits 11:8 in bytes 7:4) is described in Table 3-2. The Interrupt Stack Table (IST) field (bits 4:0 in bytes 7:4) is used by the stack switching mechanisms described in Section 6.14.5, "Interrupt Stack Table." Bytes 11:8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general-protection exception (#GP) is generated if software attempts to reference an interrupt gate with a target RIP that is not in canonical form.

The target code segment referenced by the interrupt gate must be a 64-bit code segment (CS.L = 1, CS.D = 0). If the target is not a 64-bit code segment, a general-protection exception (#GP) is generated with the IDT vector number reported as the error code.

Only 64-bit interrupt and trap gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). Legacy 32-bit interrupt or trap gate types (0EH or 0FH) are redefined in IA-32e mode as 64-bit interrupt and trap gate types. No 32-bit interrupt or trap gate type exists in IA-32e mode. If a reference is made to a 16-bit interrupt or trap gate (06H or 07H), a general-protection exception (#GP(0)) is generated.

6.14.2 64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes. SS:ESP is pushed only on a CPL change. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at eight bytes. This is because only 64-bit mode gates can be referenced. 64-bit mode also pushes SS:RSP unconditionally, rather than only on a CPL change.

When shadow stacks are enabled at the interrupt handler's privilege level and the interrupted procedure was not executing at a privilege level 3, then the processor pushes the CS:LIP:SSP of the interrupted procedure on the shadow stack of the interrupt handler (where LIP is the linear address of the return address).

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stackframe size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an additional error code place-holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA-32e mode, the RSP is aligned to a 16-byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16-byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent IRET.

Aligning the stack permits exception and interrupt frames to be aligned on a 16-byte boundary before interrupts are re-enabled. This allows the stack to be formatted for optimal storage of 16-byte XMM registers, which enables the interrupt handler to use faster 16-byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel-mode case where there is no stack switch or IST used. For a stack switch or IST, the OS would have presumably put suitably aligned RSP values in the TSS.

6.14.3 IRET in IA-32e Mode

In IA-32e mode, IRET executes with an 8-byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8-byte IRET operand size works correctly.

Because interrupt stack-frame pushes are always eight bytes in IA-32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ESP/RSP size adjustment is determined by the stack size.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL = 0. This is because the original interrupt always pushes SS:RSP.

When shadow stacks are enabled and the target privilege level is not 3, the CS:LIP from the shadow stack frame is compared to the return linear address formed by CS:EIP from the stack. If they do not match then the processor caused a control protection exception (#CP(FAR-RET/IRET)), else the processor pops the SSP of the interrupted procedure from the shadow stack. If the target privilege level is 3 and shadow stacks are enabled at privilege level 3, then the SSP for the interrupted procedure is restored from the IA32_PL3_SSP MSR.

In IA-32e mode, IRET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL \neq 3, IRET allows SS to be loaded with a NULL selector. As part of the stack switch mechanism, an interrupt or exception sets the new SS to NULL, instead of fetching a new SS selector from the TSS and loading the corresponding descriptor from the GDT or LDT. The new SS selector is set to NULL in order to properly handle returns from subsequent nested far transfers. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent IRET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

6.14.4 Stack Switching in IA-32e Mode

The IA-32 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The 64-bit extensions of Intel 64 architecture implement a modified version of the legacy stack-switching mechanism and an alternative stack-switching mechanism called the interrupt stack table (IST).

In IA-32 modes, the legacy IA-32 stack-switch mechanism is unchanged. In IA-32e mode, the legacy stack-switch mechanism is modified. When stacks are switched as part of a 64-bit mode privilege-level change (resulting from an interrupt), a new SS descriptor is not loaded. IA-32e mode loads only an inner-level RSP from the TSS. The new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INT, interrupts and exceptions). The old SS and RSP are saved on the new stack (Figure 6-9). On the subsequent IRET, the old SS is popped from the stack and loaded into the SS register.

In summary, a stack switch in IA-32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.

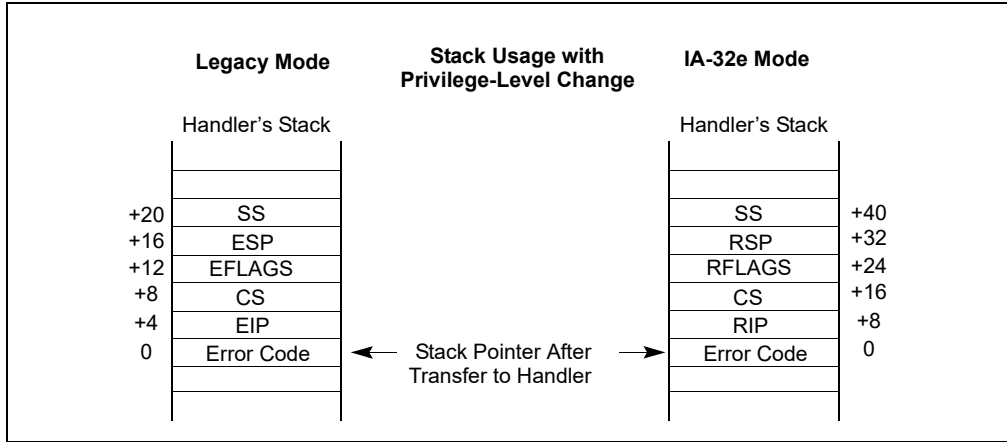


Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change

6.14.5 Interrupt Stack Table

In IA-32e mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism described above. This mechanism unconditionally switches stacks when it is enabled. It can be enabled on an individual interrupt-vector basis using a field in the IDT entry. This means that some interrupt vectors can use the modified legacy mechanism and others can use the IST mechanism.

The IST mechanism is only available in IA-32e mode. It is part of the 64-bit mode TSS. The motivation for the IST mechanism is to provide a method for specific interrupts (such as NMI, double-fault, and machine-check) to always execute on a known good stack. In legacy mode, interrupts can use the task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the legacy task-switch mechanism is not supported in IA-32e mode.

The IST mechanism provides up to seven IST pointers in the TSS. The pointers are referenced by an interrupt-gate descriptor in the interrupt-descriptor table (IDT); see Figure 6-8. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

To support this stack-switching mechanism with shadow stacks enabled, the processor provides an MSR, IA32_INTERRUPT_SSP_TABLE, to program the linear address of a table of seven shadow stack pointers that are selected using the IST index from the gate descriptor. To switch to a shadow stack selected from the interrupt shadow stack table pointed to by the IA32_INTERRUPT_SSP_TABLE, the processor requires that the shadow stack addresses programmed into this table point to a supervisor shadow stack token; see Figure 6-10.

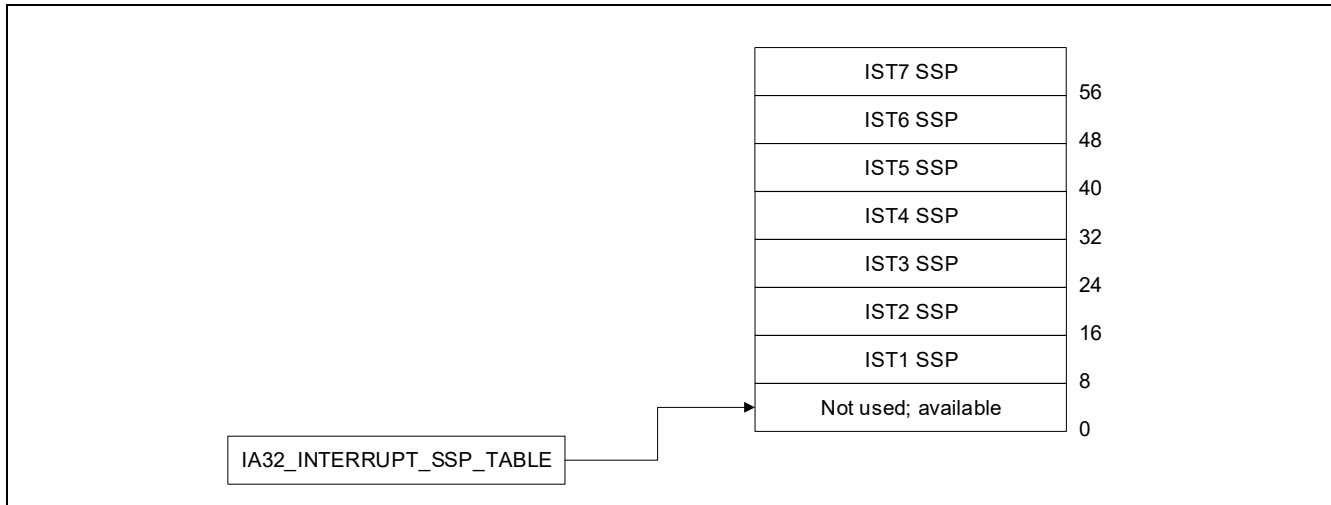


Figure 6-10. Interrupt Shadow Stack Table

6.15 EXCEPTION AND INTERRUPT REFERENCE

The following sections describe conditions which generate exceptions and interrupts. They are arranged in the order of vector numbers. The information contained in these sections are as follows:

- **Exception Class** — Indicates whether the exception class is a fault, trap, or abort type. Some exceptions can be either a fault or trap type, depending on when the error condition is detected. (This section is not applicable to interrupts.)
- **Description** — Gives a general description of the purpose of the exception or interrupt. It also describes how the processor handles the exception or interrupt.
- **Exception Error Code** — Indicates whether an error code is saved for the exception. If one is saved, the contents of the error code are described. (This section is not applicable to interrupts.)
- **Saved Instruction Pointer** — Describes which instruction the saved (or return) instruction pointer points to. It also indicates whether the pointer can be used to restart a faulting instruction.
- **Program State Change** — Describes the effects of the exception or interrupt on the state of the currently running program or task and the possibilities of restarting the program or task without loss of continuity.

Interrupt 0—Divide Error Exception (#DE)

Exception Class **Fault.**

Description

Indicates the divisor operand for a DIV or IDIV instruction is 0 or that the result cannot be represented in the number of bits specified for the destination operand.

Exception Error Code

None.

Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state change does not accompany the divide error, because the exception occurs before the faulting instruction is executed.

Interrupt 1—Debug Exception (#DB)

Exception Class **Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.**

Description

Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 6-3). See Chapter 17, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for detailed information about the debug exceptions.

Table 6-3. Debug Exception Conditions and Corresponding Exception Classes

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap
Execution of INT1 ¹	Trap

NOTES:

1. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

Saved Instruction Pointer

Fault — Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap — Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

Program State Change

Fault — A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap — A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

The following items detail the treatment of debug exceptions on the instruction boundary following execution of the MOV or the POP instruction that loads the SS register:

- If EFLAGS.TF is 1, no single-step trap is generated.
- If the instruction encounters a data breakpoint, the resulting debug exception is delivered after completion of the instruction after the MOV or POP. This occurs even if the next instruction is INT *n*, INT3, or INTO.
- Any instruction breakpoint on the instruction after the MOV or POP is suppressed (as if EFLAGS.RF were 1).

Any debug exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a debug exception instead causes execution to roll back to just before the XBEGIN instruction

and then delivers a #DB. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Interrupt 2—NMI Interrupt

Exception Class **Not applicable.**

Description

The nonmaskable interrupt (NMI) is generated externally by asserting the processor's NMI pin or through an NMI request set by the I/O APIC to the local APIC. This interrupt causes the NMI interrupt handler to be called.

Exception Error Code

Not applicable.

Saved Instruction Pointer

The processor always takes an NMI interrupt on an instruction boundary. The saved contents of CS and EIP registers point to the next instruction to be executed at the point the interrupt is taken. See Section 6.5, "Exception Classifications," for more information about when the processor takes NMI interrupts.

Program State Change

The instruction executing when an NMI interrupt is received is completed before the NMI is generated. A program or task can thus be restarted upon returning from an interrupt handler without loss of continuity, provided the interrupt handler saves the state of the processor before handling the interrupt and restores the processor's state prior to a return.

Interrupt 3—Breakpoint Exception (#BP)

Exception Class **Trap.**

Description

Indicates that a breakpoint instruction (INT3, opcode CC) was executed, causing a breakpoint trap to be generated. Typically, a debugger sets a breakpoint by replacing the first opcode byte of an instruction with the opcode for the INT3 instruction. (The INT3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode.) The operating system or a debugging tool can use a data segment mapped to the same physical address space as the code segment to place an INT3 instruction in places where it is desired to call the debugger.

With the P6 family, Pentium, Intel486, and Intel386 processors, it is more convenient to set breakpoints with the debug registers. (See Section 17.3.2, “Breakpoint Exception (#BP)—Interrupt Vector 3,” for information about the breakpoint exception.) If more breakpoints are needed beyond what the debug registers allow, the INT3 instruction can be used.

Any breakpoint exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a break exception instead causes execution to roll back to just before the XBEGIN instruction and then delivers a **debug exception (#DB)** — **not** a breakpoint exception. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

A breakpoint exception can also be generated by executing the INT *n* instruction with an operand of 3. The action of this instruction (INT 3) is slightly different than that of the INT3 instruction (see “INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

Exception Error Code

None.

Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction following the INT3 instruction.

Program State Change

Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. The debugger can thus resume the suspended program by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the saved contents of the EIP register. Upon returning from the debugger, program execution resumes with the replaced instruction.

Interrupt 4—Overflow Exception (#OF)

Exception Class **Trap.**

Description

Indicates that an overflow trap occurred when an INTO instruction was executed. The INTO instruction checks the state of the OF flag in the EFLAGS register. If the OF flag is set, an overflow trap is generated.

Some arithmetic instructions (such as the ADD and SUB) perform both signed and unsigned arithmetic. These instructions set the OF and CF flags in the EFLAGS register to indicate signed overflow and unsigned overflow, respectively. When performing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used. The benefit of using the INTO instruction is that if the overflow exception is detected, an exception handler can be called automatically to handle the overflow condition.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction following the INTO instruction.

Program State Change

Even though the EIP points to the instruction following the INTO instruction, the state of the program is essentially unchanged because the INTO instruction does not affect any register or memory locations. The program can thus resume normal execution upon returning from the overflow exception handler.

Interrupt 5—BOUND Range Exceeded Exception (#BR)

Exception Class **Fault.**

Description

Indicates that a BOUND-range-exceeded fault occurred when a BOUND instruction was executed. The BOUND instruction checks that a signed array index is within the upper and lower bounds of an array located in memory. If the array index is not within the bounds of the array, a BOUND-range-exceeded fault is generated.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the BOUND instruction that generated the exception.

Program State Change

A program-state change does not accompany the bounds-check fault, because the operands for the BOUND instruction are not modified. Returning from the BOUND-range-exceeded exception handler causes the BOUND instruction to be restarted.

Interrupt 6—Invalid Opcode Exception (#UD)

Exception Class **Fault.**

Description

Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCH h , SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCH h , SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADB, PSHUFW, PADDQ, PSUBQ, PALIGNR, PABSB, PABSD, PABSW, PHADDD, PHADDSW, PHADDW, PHSUBD, PHSUBSW, PHSUBW, PMADDUSB, PMULHRW, PSHUFB, PSIGNB, PSIGND, and PSIGNW.
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD0, UD1 or UD2 instruction. Note that even though it is the execution of the UD0, UD1 or UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still points at the UD0, UD1 or UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, "Exception Classifications," for general rules for taking of interrupts and exceptions.)

The opcodes D6 and F1 are undefined opcodes reserved by the Intel 64 and IA-32 architectures. These opcodes, even though undefined, do not generate an invalid opcode exception.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state change does not accompany an invalid-opcode fault, because the invalid instruction is not executed.

Interrupt 7—Device Not Available Exception (#NM)

Exception Class **Fault.**

Description

Indicates one of the following things:

The device-not-available exception is generated by either of three conditions:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register CR0 was set (1). See the paragraph below for the special case of the WAIT/FWAIT instruction.
- The processor executed a WAIT/FWAIT instruction while the MP and TS flags of register CR0 were set, regardless of the setting of the EM flag.
- The processor executed an x87 FPU, MMX, or SSE/SSE2/SSE3 instruction (with the exception of MOVNTI, PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, and CLFLUSH) while the TS flag in control register CR0 was set and the EM flag is clear.

The EM flag is set when the processor does not have an internal x87 FPU floating-point unit. A device-not-available exception is then generated each time an x87 FPU floating-point instruction is encountered, allowing an exception handler to call floating-point instruction emulation routines.

The TS flag indicates that a context switch (task switch) has occurred since the last time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction was executed; but that the context of the x87 FPU, XMM, and MXCSR registers were not saved. When the TS flag is set and the EM flag is clear, the processor generates a device-not-available exception each time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction is encountered (with the exception of the instructions listed above). The exception handler can then save the context of the x87 FPU, XMM, and MXCSR registers before it executes the instruction. See Section 2.5, "Control Registers," for more information about the TS flag.

The MP flag in control register CR0 is used along with the TS flag to determine if WAIT or FWAIT instructions should generate a device-not-available exception. It extends the function of the TS flag to the WAIT and FWAIT instructions, giving the exception handler an opportunity to save the context of the x87 FPU before the WAIT or FWAIT instruction is executed. The MP flag is provided primarily for use with the Intel 286 and Intel386 DX processors. For programs running on the Pentium 4, Intel Xeon, P6 family, Pentium, or Intel486 DX processors, or the Intel 487 SX coprocessors, the MP flag should always be set; for programs running on the Intel486 SX processor, the MP flag should be clear.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point instruction or the WAIT/FWAIT instruction that generated the exception.

Program State Change

A program-state change does not accompany a device-not-available fault, because the instruction that generated the exception is not executed.

If the EM flag is set, the exception handler can then read the floating-point instruction pointed to by the EIP and call the appropriate emulation routine.

If the MP and TS flags are set or the TS flag alone is set, the exception handler can save the context of the x87 FPU, clear the TS flag, and continue execution at the interrupted floating-point or WAIT/FWAIT instruction.

Interrupt 8—Double Fault Exception (#DF)

Exception Class **Abort.**

Description

Indicates that the processor detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception. To determine when two faults need to be signalled as a double fault, the processor divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults (see Table 6-4).

Table 6-4. Interrupt and Exception Classes

Class	Vector Number	Description
Benign Exceptions and Interrupts	1	Debug
	2	NMI Interrupt
	3	Breakpoint
	4	Overflow
	5	BOUND Range Exceeded
	6	Invalid Opcode
	7	Device Not Available
	9	Coprocessor Segment Overrun
	16	Floating-Point Error
	17	Alignment Check
	18	Machine Check
	19	SIMD floating-point
	All	INT <i>n</i>
All	INTR	
Contributory Exceptions	0	Divide Error
	10	Invalid TSS
	11	Segment Not Present
	12	Stack Fault
	13	General Protection
	21	Control Protection
Page Faults	14	Page Fault
	20	Virtualization Exception

Table 6-5 shows the various combinations of exception classes that cause a double fault to be generated. A double-fault exception falls in the abort class of exceptions. The program or task cannot be restarted or resumed. The double-fault handler can be used to collect diagnostic information about the state of the machine and/or, when possible, to shut the application and/or system down gracefully or restart the system.

A segment or page fault may be encountered while prefetching instructions; however, this behavior is outside the domain of Table 6-5. Any further faults generated while the processor is attempting to transfer control to the appropriate fault handler could still lead to a double-fault sequence.

Table 6-5. Conditions for Generating a Double Fault

First Exception	Second Exception		
	Benign	Contributory	Page Fault
Benign	Handle Exceptions Serially	Handle Exceptions Serially	Handle Exceptions Serially
Contributory	Handle Exceptions Serially	Generate a Double Fault	Handle Exceptions Serially
Page Fault	Handle Exceptions Serially	Generate a Double Fault	Generate a Double Fault
Double Fault	Handle Exceptions Serially	Enter Shutdown Mode	Enter Shutdown Mode

If another contributory or page fault exception occurs while attempting to call the double-fault handler, the processor enters shutdown mode. This mode is similar to the state following execution of an HLT instruction. In this mode, the processor stops executing instructions until an NMI interrupt, SMI interrupt, hardware reset, or INIT# is received. The processor generates a special bus cycle to indicate that it has entered shutdown mode. Software designers may need to be aware of the response of hardware when it goes into shutdown mode. For example, hardware may turn on an indicator light on the front panel, generate an NMI interrupt to record diagnostic information, invoke reset initialization, generate an INIT initialization, or generate an SMI. If any events are pending during shutdown, they will be handled after a wake event from shutdown is processed (for example, A20M# interrupts).

If a shutdown occurs while the processor is executing an NMI interrupt handler, then only a hardware reset can restart the processor. Likewise, if the shutdown occurs while executing in SMM, a hardware reset must be used to restart the processor.

Exception Error Code

Zero. The processor always pushes an error code of 0 onto the stack of the double-fault handler.

Saved Instruction Pointer

The saved contents of CS and EIP registers are undefined.

Program State Change

A program-state following a double-fault exception is undefined. The program or task cannot be resumed or restarted. The only available action of the double-fault exception handler is to collect all possible context information for use in diagnostics and then close the application and/or shut down or reset the processor.

If the double fault occurs when any portion of the exception handling machine state is corrupted, the handler cannot be invoked and the processor must be reset.

Interrupt 9—Coprocessor Segment Overrun

Exception Class **Abort. (Intel reserved; do not use. Recent IA-32 processors do not generate this exception.)**

Description

Indicates that an Intel386 CPU-based systems with an Intel 387 math coprocessor detected a page or segment violation while transferring the middle portion of an Intel 387 math coprocessor operand. The P6 family, Pentium, and Intel486 processors do not generate this exception; instead, this condition is detected with a general protection exception (`#GP`), interrupt 13.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state following a coprocessor segment-overrun exception is undefined. The program or task cannot be resumed or restarted. The only available action of the exception handler is to save the instruction pointer and reinitialize the x87 FPU using the `FNINIT` instruction.

Interrupt 10—Invalid TSS Exception (#TS)

Exception Class **Fault.**

Description

Indicates that there was an error related to a TSS. Such an error might be detected during a task switch or during the execution of instructions that use information from a TSS. Table 6-6 shows the conditions that cause an invalid TSS exception to be generated.

Table 6-6. Invalid TSS Conditions

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment limit is less than 67H for 32-bit TSS or less than 2CH for 16-bit TSS.
TSS segment selector index	During an IRET task switch, the TI flag in the TSS segment selector indicates the LDT.
TSS segment selector index	During an IRET task switch, the TSS segment selector exceeds descriptor table limit.
TSS segment selector index	During an IRET task switch, the busy flag in the TSS descriptor indicates an inactive task.
TSS segment selector index	During a task switch, an attempt to access data in a TSS results in a limit violation or canonical fault.
TSS segment selector index	During an IRET task switch, the backlink is a NULL selector.
TSS segment selector index	During an IRET task switch, the backlink points to a descriptor which is not a busy TSS.
TSS segment selector index	The new TSS descriptor is beyond the GDT limit.
TSS segment selector index	The new TSS selector is null on an attempt to lock the new TSS.
TSS segment selector index	The new TSS selector has the TI bit set on an attempt to lock the new TSS.
TSS segment selector index	The new TSS descriptor is not an available TSS descriptor on an attempt to lock the new TSS.
LDT segment selector index	LDT not valid or not present.
Stack segment selector index	The stack segment selector exceeds descriptor table limit.
Stack segment selector index	The stack segment selector is NULL.
Stack segment selector index	The stack segment descriptor is a non-data segment.
Stack segment selector index	The stack segment is not writable.
Stack segment selector index	The stack segment DPL ≠ CPL.
Stack segment selector index	The stack segment selector RPL ≠ CPL.
Code segment selector index	The code segment selector exceeds descriptor table limit.
Code segment selector index	The code segment selector is NULL.
Code segment selector index	The code segment descriptor is not a code segment type.
Code segment selector index	The nonconforming code segment DPL ≠ CPL.
Code segment selector index	The conforming code segment DPL is greater than CPL.
Data segment selector index	The data segment selector exceeds the descriptor table limit.
Data segment selector index	The data segment descriptor is not a readable code or data type.
Data segment selector index	The data segment descriptor is a nonconforming code type and RPL > DPL.
Data segment selector index	The data segment descriptor is a nonconforming code type and CPL > DPL.
TSS segment selector index	The TSS segment descriptor/upper descriptor is beyond the GDT segment limit.
TSS segment selector index	The TSS segment descriptor is not an available TSS type.
TSS segment selector index	The TSS segment descriptor is an available 286 TSS type in IA-32e mode.

Table 6-6. Invalid TSS Conditions (Contd.)

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment upper descriptor is not the correct type.
TSS segment selector index	The TSS segment descriptor contains a non-canonical base.

This exception can be generated either in the context of the original task or in the context of the new task (see Section 7.3, "Task Switching"). Until the processor has completely verified the presence of the new TSS, the exception is generated in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete. Any invalid-TSS conditions detected after this point are handled in the context of the new task. (A task switch is considered complete when the task register is loaded with the segment selector for the new TSS and, if the switch is due to a procedure call or interrupt, the previous task link field of the new TSS references the old TSS.)

The invalid-TSS handler must be a task called using a task gate. Handling this exception inside the faulting TSS context is not recommended because the processor state may not be consistent.

Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception was caused by an event external to the currently running program (for example, if an external interrupt handler using a task gate attempted a task switch to an invalid TSS).

Saved Instruction Pointer

If the exception condition was detected before the task switch was carried out, the saved contents of CS and EIP registers point to the instruction that invoked the task switch. If the exception condition was detected after the task switch was carried out, the saved contents of CS and EIP registers point to the first instruction of the new task.

Program State Change

The ability of the invalid-TSS handler to recover from the fault depends on the error condition that causes the fault. See Section 7.3, "Task Switching," for more information on the task switch process and the possible recovery actions that can be taken.

If an invalid TSS exception occurs during a task switch, it can occur before or after the commit-to-new-task point. If it occurs before the commit point, no program state change occurs. If it occurs after the commit point (when the segment descriptor information for the new segment selectors have been loaded in the segment registers), the processor will load all the state information from the new TSS before it generates the exception. During a task switch, the processor first loads all the segment registers with segment selectors from the TSS, then checks their contents for validity. If an invalid TSS exception is discovered, the remaining segment registers are loaded but not checked for validity and therefore may not be usable for referencing memory. The invalid TSS handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should load all segment registers before trying to resume the new task; otherwise, general-protection exceptions (#GP) may result later under conditions that make diagnosis more difficult. The Intel recommended way of dealing with this situation is to use a task for the invalid TSS exception handler. The task switch back to the interrupted task from the invalid-TSS exception-handler task will then cause the processor to check the registers as it loads them from the TSS.

Interrupt 11—Segment Not Present (#NP)

Exception Class **Fault.**

Description

Indicates that the present flag of a segment or gate descriptor is clear. The processor can generate this exception during any of the following operations:

- While attempting to load CS, DS, ES, FS, or GS registers. [Detection of a not-present segment while loading the SS register causes a stack fault exception (#SS) to be generated.] This situation can occur while performing a task switch.
- While attempting to load the LDTR using an LLDT instruction. Detection of a not-present LDT while loading the LDTR during a task switch operation causes an invalid-TSS exception (#TS) to be generated.
- When executing the LTR instruction and the TSS is marked not present.
- While attempting to use a gate descriptor or TSS that is marked segment-not-present, but is otherwise valid.

An operating system typically uses the segment-not-present exception to implement virtual memory at the segment level. If the exception handler loads the segment and returns, the interrupted program or task resumes execution.

A not-present indication in a gate descriptor, however, does not indicate that a segment is not present (because gates do not correspond to segments). The operating system may use the present flag for gate descriptors to trigger exceptions of special significance to the operating system.

A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

- an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment
- a benign exception that subsequently referenced a not-present segment

The IDT flag is set if the error code refers to an IDT entry. This occurs when the IDT entry for an interrupt being serviced references a not-present gate. Such an event could be generated by an INT instruction or a hardware interrupt.

Saved Instruction Pointer

The saved contents of CS and EIP registers normally point to the instruction that generated the exception. If the exception occurred while loading segment descriptors for the segment selectors in a new TSS, the CS and EIP registers point to the first instruction in the new task. If the exception occurred while accessing a gate descriptor, the CS and EIP registers point to the instruction that invoked the access (for example a CALL instruction that references a call gate).

Program State Change

If the segment-not-present exception occurs as the result of loading a register (CS, DS, SS, ES, FS, GS, or LDTR), a program-state change does accompany the exception because the register is not loaded. Recovery from this exception is possible by simply loading the missing segment into memory and setting the present flag in the segment descriptor.

If the segment-not-present exception occurs while accessing a gate descriptor, a program-state change does not accompany the exception. Recovery from this exception is possible merely by setting the present flag in the gate descriptor.

If a segment-not-present exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change

occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The segment-not-present exception handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

Interrupt 12—Stack Fault Exception (#SS)

Exception Class **Fault.**

Description

Indicates that one of the following stack related conditions was detected:

- A limit violation is detected during an operation that refers to the SS register. Operations that can cause a limit violation include stack-oriented instructions such as POP, PUSH, CALL, RET, IRET, ENTER, and LEAVE, as well as other memory references which implicitly or explicitly use the SS register (for example, MOV AX, [BP+6] or MOV AX, SS:[EAX+6]). The ENTER instruction generates this exception when there is not enough stack space for allocating local variables.
- A not-present stack segment is detected when attempting to load the SS register. This violation can occur during the execution of a task switch, a CALL instruction to a different privilege level, a return to a different privilege level, an LSS instruction, or a MOV or POP instruction to the SS register.
- A canonical violation is detected in 64-bit mode during an operation that reference memory using the stack pointer register containing a non-canonical memory address.

Recovery from this fault is possible by either extending the limit of the stack segment (in the case of a limit violation) or loading the missing stack segment into memory (in the case of a not-present violation).

In the case of a canonical violation that was caused intentionally by software, recovery is possible by loading the correct canonical value into RSP. Otherwise, a canonical violation of the address in RSP likely reflects some register corruption in the software.

Exception Error Code

If the exception is caused by a not-present stack segment or by overflow of the new stack during an inter-privilege-level call, the error code contains a segment selector for the segment that caused the exception. Here, the exception handler can test the present flag in the segment descriptor pointed to by the segment selector to determine the cause of the exception. For a normal limit violation (on a stack segment already in use) the error code is set to 0.

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. However, when the exception results from attempting to load a not-present stack segment during a task switch, the CS and EIP registers point to the first instruction of the new task.

Program State Change

A program-state change does not generally accompany a stack-fault exception, because the instruction that generated the fault is not executed. Here, the instruction can be restarted after the exception handler has corrected the stack fault condition.

If a stack fault occurs during a task switch, it occurs after the commit-to-new-task point (see Section 7.3, "Task Switching"). Here, the processor loads all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The stack fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that are more difficult to diagnose. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

Interrupt 13—General Protection Exception (#GP)

Exception Class **Fault.**

Description

Indicates that the processor detected one of a class of protection violations called “general-protection violations.” The conditions that cause this exception to be generated comprise all the protection violations that do not cause other exceptions to be generated (such as, invalid-TSS, segment-not-present, stack-fault, or page-fault exceptions). The following conditions cause general-protection exceptions to be generated:

- Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
- Transferring execution to a segment that is not executable.
- Writing to a code segment or a read-only data segment.
- Reading from an execute-only code segment.
- Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment.
- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, “Protection.”
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler’s code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, “Privileged Instructions,” for a list of privileged instructions).
- Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT *n* instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.

- If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1. These bits are checked during a write to control registers CR0, CR3, or CR4 that causes a reloading of the page-directory-pointer-table entry.
- Attempting to write a non-zero value into the reserved bits of the MXCSR register.
- Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.

A program or task can be restarted following any general-protection exception. If the exception occurs while attempting to call an interrupt handler, the interrupted program can be restartable, but the interrupt may be lost.

Exception Error Code

The processor pushes an error code onto the exception handler's stack. If the fault condition was detected while loading a segment descriptor, the error code contains a segment selector to or IDT vector number for the descriptor; otherwise, the error code is 0. The source of the selector in an error code may be any of the following:

- An operand of the instruction.
- A selector from a gate which is the operand of the instruction.
- A selector from a TSS involved in a task switch.
- IDT vector number.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

In general, a program-state change does not accompany a general-protection exception, because the invalid instruction or operation is not executed. An exception handler can be designed to correct all of the conditions that cause general-protection exceptions and restart the program or task without any loss of program continuity.

If a general-protection exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The general-protection exception handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

General Protection Exception in 64-bit Mode

The following conditions cause general-protection exceptions in 64-bit mode:

- If the memory address is in a non-canonical form.
- If a segment descriptor memory address is in non-canonical form.
- If the target offset in a destination operand of a call or jmp is in a non-canonical form.
- If a code segment or 64-bit call gate overlaps non-canonical space.
- 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 EFLAGS.NT bit is set in IRET.
- If the stack segment selector of IRET is null when going back to compatibility mode.
- If the stack segment selector of IRET is null going back to CPL3 and 64-bit mode.
- If a null stack segment selector RPL of IRET is not equal to CPL going back to non-CPL3 and 64-bit mode.
- If the proposed new code segment descriptor of IRET has both the D-bit and the L-bit set.

- If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and it has both the D-bit and the L-bit set.
- If the segment descriptor from a 64-bit call gate is in non-canonical space.
- 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 type field of the upper 64 bits of a 64-bit call gate is not 0.
- If an attempt is made to load a null selector in the SS register in compatibility mode.
- If an attempt is made to load null selector in the SS register in CPL3 and 64-bit mode.
- If an attempt is made to load a null selector in the SS register in non-CPL3 and 64-bit mode where RPL is not equal to CPL.
- If an attempt is made to clear CR0.PG while IA-32e mode is enabled.
- If an attempt is made to set a reserved bit in CR3, CR4 or CR8.

Interrupt 14—Page-Fault Exception (#PF)

Exception Class **Fault.**

Description

Indicates that, with paging enabled (the PG flag in the CR0 register is set), the processor detected one of the following conditions while using the page-translation mechanism to translate a linear address to a physical address:

- The P (present) flag in a page-directory or page-table entry needed for the address translation is clear, indicating that a page table or the page containing the operand is not present in physical memory.
- The procedure does not have sufficient privilege to access the indicated page (that is, a procedure running in user mode attempts to access a supervisor-mode page). If the SMAP flag is set in CR4, a page fault may also be triggered by code running in supervisor mode that tries to access data at a user-mode address. If **either** the PKE flag **or the** PKS flag is set in CR4, the **protection-key rights** registers may cause page faults on data accesses to **linear** addresses with certain protection keys.
- Code running in user mode attempts to write to a read-only page. If the WP flag is set in CR0, the page fault will also be triggered by code running in supervisor mode that tries to write to a read-only page.
- An instruction fetch to a linear address that translates to a physical address in a memory page with the execute-disable bit set (for information about the execute-disable bit, see Chapter 4, “Paging”). If the SMEP flag is set in CR4, a page fault will also be triggered by code running in supervisor mode that tries to fetch an instruction from a user-mode address.
- One or more reserved bits in paging-structure entry are set to 1. See description below of RSVD error code flag.
- A shadow-stack access is made to a page that is not a shadow-stack page. See Section 18.2, “Shadow Stacks” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* and Chapter 4.6, “Access Rights.”
- An enclave access violates one of the specified access-control requirements. See Section 37.3, “Access-control Requirements” and Section 37.20, “Enclave Page Cache Map (EPCM)” in Chapter 37, “Enclave Access Control and Data Structures.” In this case, the exception is called an **SGX-induced page fault**. The processor uses the error code (below) to distinguish SGX-induced page faults from ordinary page faults.

The exception handler can recover from page-not-present conditions and restart the program or task without any loss of program continuity. It can also restart the program or task after a privilege violation, but the problem that caused the privilege violation may be uncorrectable.

See also: Section 4.7, “Page-Fault Exceptions.”

Exception Error Code

Yes (special format). The processor provides the page-fault handler with two items of information to aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 6-11). The processor establishes the bits in the error code as follows:
 - 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.

- 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.
- I/D flag (bit 4).
This flag is 1 if the access causing the page-fault exception was an instruction fetch. 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 the access causing the page-fault exception was a data access to a linear address with a protection key for which the protection-key rights registers disallow access.
- SS (bit 1).
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), 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.
- 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.

See Section 4.6, “Access Rights” and Section 4.7, “Page-Fault Exceptions” for more information about page-fault exceptions and the error codes that they produce.

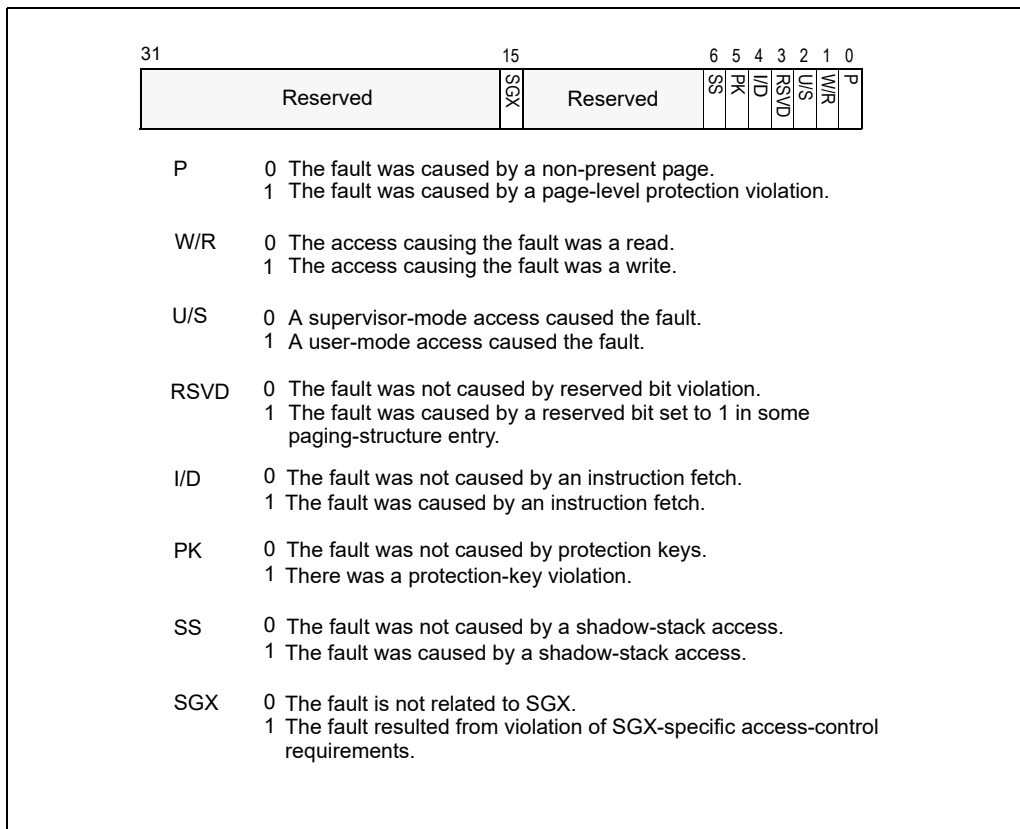


Figure 6-11. Page-Fault Error Code

- The contents of the CR2 register. The processor loads the CR2 register with the 32-bit linear address that generated the exception. The page-fault handler can use this address to locate the corresponding page directory and page-table entries. Another page fault can potentially occur during execution of the page-fault handler; the handler should save the contents of the CR2 register before a second page fault can occur.⁶ If a page fault is caused by a page-level protection violation, the access flag in the page-directory entry is set when

the fault occurs. The behavior of IA-32 processors regarding the access flag in the corresponding page-table entry is model specific and not architecturally defined.

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. If the page-fault exception occurred during a task switch, the CS and EIP registers may point to the first instruction of the new task (as described in the following “Program State Change” section).

Program State Change

A program-state change does not normally accompany a page-fault exception, because the instruction that causes the exception to be generated is not executed. After the page-fault exception handler has corrected the violation (for example, loaded the missing page into memory), execution of the program or task can be resumed.

When a page-fault exception is generated during a task switch, the program-state may change, as follows. During a task switch, a page-fault exception can occur during any of following operations:

- While writing the state of the original task into the TSS of that task.
- While reading the GDT to locate the TSS descriptor of the new task.
- While reading the TSS of the new task.
- While reading segment descriptors associated with segment selectors from the new task.
- While reading the LDT of the new task to verify the segment registers stored in the new TSS.

In the last two cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be called through a task gate.

If a page fault occurs during a task switch, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The page-fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

Additional Exception-Handling Information

Special care should be taken to ensure that an exception that occurs during an explicit stack switch does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for 16-bit IA-32 processors often use a pair of instructions to change to a new stack, for example:

```
MOV SS, AX
MOV SP, StackTop
```

When executing this code on one of the 32-bit IA-32 processors, it is possible to get a page fault, general-protection fault (#GP), or alignment check fault (#AC) after the segment selector has been loaded into the SS register but before the ESP register has been loaded. At this point, the two parts of the stack pointer (SS and ESP) are inconsistent. The new stack segment is being used with the old stack pointer.

The processor does not use the inconsistent stack pointer if the exception handler switches to a well defined stack (that is, the handler is a task or a more privileged procedure). However, if the exception handler is called at the same privilege level and from the same task, the processor will attempt to use the inconsistent stack pointer.

In systems that handle page-fault, general-protection, or alignment check exceptions within the faulting task (with trap or interrupt gates), software executing at the same privilege level as the exception handler should initialize a new stack by using the LSS instruction rather than a pair of MOV instructions, as described earlier in this note. When the exception handler is running at privilege level 0 (the normal case), the problem is limited to procedures or tasks that run at privilege level 0, typically the kernel of the operating system.

-
6. Processors update CR2 whenever a page fault is detected. If a second page fault occurs while an earlier page fault is being delivered, the faulting linear address of the second fault will overwrite the contents of CR2 (replacing the previous address). These updates to CR2 occur even if the page fault results in a double fault or occurs during the delivery of a double fault.

Interrupt 16—x87 FPU Floating-Point Error (#MF)

Exception Class **Fault.**

Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, “Control Registers,” for a detailed description of the NE flag.)

NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

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

Each of these error conditions represents an x87 FPU exception type, and for each of exception type, the x87 FPU provides a flag in the x87 FPU status register and a mask bit in the x87 FPU control register. If the x87 FPU detects a floating-point error and the mask bit for the exception type is set, the x87 FPU handles the exception automatically by generating a predefined (default) response and continuing program execution. The default responses have been designed to provide a reasonable result for most floating-point applications.

If the mask for the exception is clear and the NE flag in register CR0 is set, the x87 FPU does the following:

1. Sets the necessary flag in the FPU status register.
2. Waits until the next “waiting” x87 FPU instruction or WAIT/FWAIT instruction is encountered in the program’s instruction stream.
3. Generates an internal error signal that cause the processor to generate a floating-point exception (#MF).

Prior to executing a waiting x87 FPU instruction or the WAIT/FWAIT instruction, the x87 FPU checks for pending x87 FPU floating-point exceptions (as described in step 2 above). Pending x87 FPU floating-point exceptions are ignored for “non-waiting” x87 FPU instructions, which include the FNINIT, FNCLEX, FNSTSW, FNSTSW AX, FNSTCW, FNSTENV, and FNSAVE instructions. Pending x87 FPU exceptions are also ignored when executing the state management instructions FXSAVE and FXRSTOR.

All of the x87 FPU floating-point error conditions can be recovered from. The x87 FPU floating-point-error exception handler can determine the error condition that caused the exception from the settings of the flags in the x87 FPU status word. See “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on handling x87 FPU floating-point exceptions.

Exception Error Code

None. The x87 FPU provides its own error information.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point or WAIT/FWAIT instruction that was about to be executed when the floating-point-error exception was generated. This is not the faulting instruction in which the error condition was detected. The address of the faulting instruction is contained in the x87 FPU instruction pointer

register. See Section 8.1.8, “x87 FPU Instruction and Data (Operand) Pointers” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about information the FPU saves for use in handling floating-point-error exceptions.

Program State Change

A program-state change generally accompanies an x87 FPU floating-point exception because the handling of the exception is delayed until the next waiting x87 FPU floating-point or WAIT/FWAIT instruction following the faulting instruction. The x87 FPU, however, saves sufficient information about the error condition to allow recovery from the error and re-execution of the faulting instruction if needed.

In situations where non- x87 FPU floating-point instructions depend on the results of an x87 FPU floating-point instruction, a WAIT or FWAIT instruction can be inserted in front of a dependent instruction to force a pending x87 FPU floating-point exception to be handled before the dependent instruction is executed. See “x87 FPU Exception Synchronization” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about synchronization of x87 floating-point-error exceptions.

Interrupt 17—Alignment Check Exception (#AC)

Exception Class **Fault.**

Description

Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) accesses (not in code fetches or system segment accesses). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 6-7 lists the alignment requirements various data types recognized by the processor.

Table 6-7. Alignment Requirements by Data Type

Data Type	Address Must Be Divisible By
Word	2
Doubleword	4
Single-precision floating-point (32-bits)	4
Double-precision floating-point (64-bits)	8
Double extended-precision floating-point (80-bits)	8
Quadword	8
Double quadword	16
Segment Selector	2
32-bit Far Pointer	2
48-bit Far Pointer	4
32-bit Pointer	4
GDTR, IDTR, LDTR, or Task Register Contents	4
FSTENV/FLDENV Save Area	4 or 2, depending on operand size
FSAVE/FRSTOR Save Area	4 or 2, depending on operand size
Bit String	2 or 4 depending on the operand-size attribute.

Note that the alignment check exception (#AC) is generated only for data types that must be aligned on word, doubleword, and quadword boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:

- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (including virtual-8086 mode).

Alignment-check exceptions (#AC) are generated only when operating at privilege level 3 (user mode). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate alignment-check exceptions, even when caused by a memory reference made from privilege level 3.

Storing the contents of the GDTR, IDTR, LDTR, or task register in memory while at privilege level 3 can generate an alignment-check exception. Although application programs do not normally store these registers, the fault can be avoided by aligning the information stored on an even word-address.

The FXSAVE/XSAVE and FXRSTOR/XRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the processor implementation (see “FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State” and “FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State” in

Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*; see "XSAVE—Save Processor Extended States" and "XRSTOR—Restore Processor Extended States" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*).

The MOVDQU, MOVUPS, and MOVUPD instructions perform 128-bit unaligned loads or stores. The LDDQU instruction loads 128-bit unaligned data. They do not generate general-protection exceptions (#GP) when operands are not aligned on a 16-byte boundary. If alignment checking is enabled, alignment-check exceptions (#AC) may or may not be generated depending on processor implementation when data addresses are not aligned on an 8-byte boundary.

FSAVE and FRSTOR instructions can generate unaligned references, which can cause alignment-check faults. These instructions are rarely needed by application programs.

Exception Error Code

Yes. The error code is null; all bits are clear except possibly bit 0 — EXT; see Section 6.13. EXT is set if the #AC is recognized during delivery of an event other than a software interrupt (see "INT n/INTO/INT3/INT1—Call to Interrupt Procedure" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state change does not accompany an alignment-check fault, because the instruction is not executed.

Interrupt 18—Machine-Check Exception (#MC)

Exception Class **Abort.**

Description

Indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The machine-check exception is model-specific, available on the Pentium and later generations of processors. The implementation of the machine-check exception is different between different processor families, and these implementations may not be compatible with future Intel 64 or IA-32 processors. (Use the CPUID instruction to determine whether this feature is present.)

Bus errors detected by external agents are signaled to the processor on dedicated pins: the BINIT# and MCERR# pins on the Pentium 4, Intel Xeon, and P6 family processors and the BUSCHK# pin on the Pentium processor. When one of these pins is enabled, asserting the pin causes error information to be loaded into machine-check registers and a machine-check exception is generated.

The machine-check exception and machine-check architecture are discussed in detail in Chapter 15, “Machine-Check Architecture.” Also, see the data books for the individual processors for processor-specific hardware information.

Exception Error Code

None. Error information is provided by machine-check MSRs.

Saved Instruction Pointer

For the Pentium 4 and Intel Xeon processors, the saved contents of extended machine-check state registers are directly associated with the error that caused the machine-check exception to be generated (see Section 15.3.1.2, “IA32_MCG_STATUS MSR,” and Section 15.3.2.6, “IA32_MCG Extended Machine Check State MSRs”).

For the P6 family processors, if the EIPV flag in the MCG_STATUS MSR is set, the saved contents of CS and EIP registers are directly associated with the error that caused the machine-check exception to be generated; if the flag is clear, the saved instruction pointer may not be associated with the error (see Section 15.3.1.2, “IA32_MCG_STATUS MSR”).

For the Pentium processor, contents of the CS and EIP registers may not be associated with the error.

Program State Change

The machine-check mechanism is enabled by setting the MCE flag in control register CR4.

For the Pentium 4, Intel Xeon, P6 family, and Pentium processors, a program-state change always accompanies a machine-check exception, and an abort class exception is generated. For abort exceptions, information about the exception can be collected from the machine-check MSRs, but the program cannot generally be restarted.

If the machine-check mechanism is not enabled (the MCE flag in control register CR4 is clear), a machine-check exception causes the processor to enter the shutdown state.

Interrupt 19—SIMD Floating-Point Exception (#XM)

Exception Class **Fault.**

Description

Indicates the processor has detected an SSE/SSE2/SSE3 SIMD floating-point exception. The appropriate status flag in the MXCSR register must be set and the particular exception unmasked for this interrupt to be generated.

There are six classes of numeric exception conditions that can occur while executing an SSE/ SSE2/SSE3 SIMD floating-point instruction:

- Invalid operation (#I)
- Divide-by-zero (#Z)
- Denormal operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (Precision) (#P)

The invalid operation, divide-by-zero, and denormal-operand exceptions are pre-computation exceptions; that is, they are detected before any arithmetic operation occurs. The numeric underflow, numeric overflow, and inexact result exceptions are post-computational exceptions.

See “SIMD Floating-Point Exceptions” in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information about the SIMD floating-point exception classes.

When a SIMD floating-point exception occurs, the processor does either of the following things:

- It handles the exception automatically by producing the most reasonable result and allowing program execution to continue undisturbed. This is the response to masked exceptions.
- It generates a SIMD floating-point exception, which in turn invokes a software exception handler. This is the response to unmasked exceptions.

Each of the six SIMD floating-point exception conditions has a corresponding flag bit and mask bit in the MXCSR register. If an exception is masked (the corresponding mask bit in the MXCSR register is set), the processor takes an appropriate automatic default action and continues with the computation. If the exception is unmasked (the corresponding mask bit is clear) and the operating system supports SIMD floating-point exceptions (the OSXM-MEXCPT flag in control register CR4 is set), a software exception handler is invoked through a SIMD floating-point exception. If the exception is unmasked and the OSXMMEXCPT bit is clear (indicating that the operating system does not support unmasked SIMD floating-point exceptions), an invalid opcode exception (#UD) is signaled instead of a SIMD floating-point exception.

Note that because SIMD floating-point exceptions are precise and occur immediately, the situation does not arise where an x87 FPU instruction, a WAIT/FWAIT instruction, or another SSE/SSE2/SSE3 instruction will catch a pending unmasked SIMD floating-point exception.

In situations where a SIMD floating-point exception occurred while the SIMD floating-point exceptions were masked (causing the corresponding exception flag to be set) and the SIMD floating-point exception was subsequently unmasked, then no exception is generated when the exception is unmasked.

When SSE/SSE2/SSE3 SIMD floating-point instructions operate on packed operands (made up of two or four sub-operands), multiple SIMD floating-point exception conditions may be detected. If no more than one exception condition is detected for one or more sets of sub-operands, the exception flags are set for each exception condition detected. For example, an invalid exception detected for one sub-operand will not prevent the reporting of a divide-by-zero exception for another sub-operand. However, when two or more exceptions conditions are generated for one sub-operand, only one exception condition is reported, according to the precedences shown in Table 6-8. This exception precedence sometimes results in the higher priority exception condition being reported and the lower priority exception conditions being ignored.

Table 6-8. SIMD Floating-Point Exceptions Priority

Priority	Description
1 (Highest)	Invalid operation exception due to SNaN operand (or any NaN operand for maximum, minimum, or certain compare and convert operations).
2	QNaN operand ¹ .
3	Any other invalid operation exception not mentioned above or a divide-by-zero exception ² .
4	Denormal operand exception ² .
5	Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception ² .
6 (Lowest)	Inexact result exception.

NOTES:

1. Though a QNaN this is not an exception, the handling of a QNaN operand has precedence over lower priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a divide-by-zero- exception.
2. If masked, then instruction execution continues, and a lower priority exception can occur as well.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the SSE/SSE2/SSE3 instruction that was executed when the SIMD floating-point exception was generated. This is the faulting instruction in which the error condition was detected.

Program State Change

A program-state change does not accompany a SIMD floating-point exception because the handling of the exception is immediate unless the particular exception is masked. The available state information is often sufficient to allow recovery from the error and re-execution of the faulting instruction if needed.

Interrupt 20—Virtualization Exception (#VE)

Exception Class **Fault.**

Description

Indicates that the processor detected an EPT violation in VMX non-root operation. Not all EPT violations cause virtualization exceptions. See Section 25.5.7.2 for details.

The exception handler can recover from EPT violations and restart the program or task without any loss of program continuity. In some cases, however, the problem that caused the EPT violation may be uncorrectable.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception.

Program State Change

A program-state change does not normally accompany a virtualization exception, because the instruction that causes the exception to be generated is not executed. After the virtualization exception handler has corrected the violation (for example, by executing the EPTP-switching VM function), execution of the program or task can be resumed.

Additional Exception-Handling Information

The processor saves information about virtualization exceptions in the virtualization-exception information area. See Section 25.5.7.2 for details.

Interrupt 21—Control Protection Exception (#CP)

Exception Class **Fault.**

Description

Indicates a control flow transfer attempt violated the control flow enforcement technology constraints.

Exception Error Code

Yes (special format). The processor provides the control protection exception handler with following information through the error code on the stack.

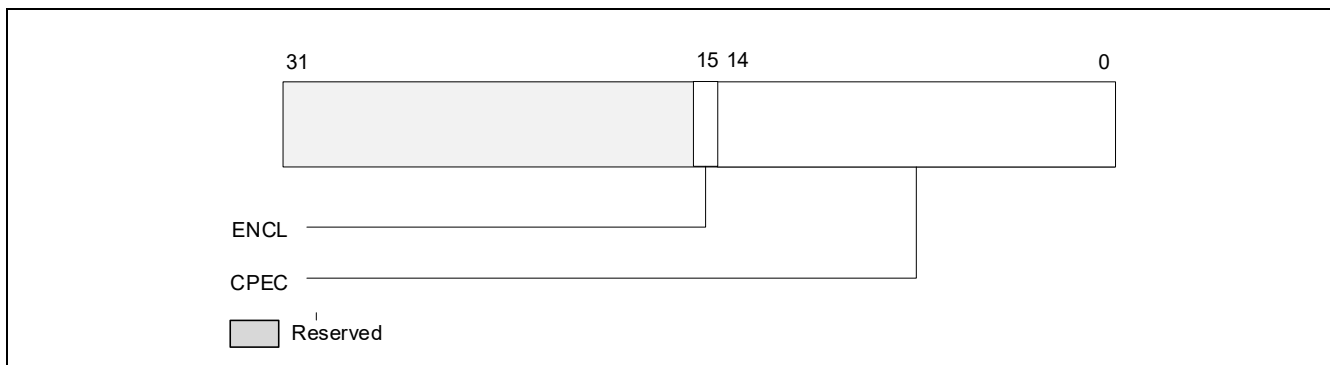


Figure 6-12. Exception Error Code Information

- Bit 14:0 - CPEC
 - 1 - NEAR-RET: Indicates the #CP was caused by a near RET instruction.
 - 2 - FAR-RET/IRET: Indicates the #CP was caused by a FAR RET or IRET instruction.
 - 3 - ENDBRANCH: indicates the #CP was due to missing ENDBRANCH at target of an indirect call or jump instruction.
 - 4 - RSTORSSP: Indicates the #CP was caused by a shadow-stack-restore token check failure in the RSTORSSP instruction.
 - 5- SETSSBSY: Indicates #CP was caused by a supervisor shadow stack token check failure in the SETSSBSY instruction.
- Bit 15 (ENCL) of the error code, if set to 1, indicates the #CP occurred during enclave execution.

Saved Instruction Pointer

The saved contents of the CS and EIP registers generally point to the instruction that generated the exception.

Program State Change

A program-state change does not normally accompany a control protection exception, because the instruction that causes the exception to be generated is not executed.

When a control protection exception is generated during a task switch, the program-state may change as follows. During a task switch, a control protection exception can occur during any of following operations:

- If task switch is initiated by IRET, CS and LIP stored on old task shadow stack do not match CS and LIP of new task (where LIP is the linear address of the return address).
- If task switch is initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3), OR the SSP from IA32_PL3_SSP (if new task CPL = 3) is not aligned to 4 bytes or is a value beyond 4GB.

INTERRUPT AND EXCEPTION HANDLING

In these cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits control protection faults to occur during task-switches, the control protection fault handler should be called through a task gate.

Interrupts 32 to 255—User Defined Interrupts

Exception Class **Not applicable.**

Description

Indicates that the processor did one of the following things:

- Executed an `INT n` instruction where the instruction operand is one of the vector numbers from 32 through 255.
- Responded to an interrupt request at the INTR pin or from the local APIC when the interrupt vector number associated with the request is from 32 through 255.

Exception Error Code

Not applicable.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that follows the `INT n` instruction or instruction following the instruction on which the INTR signal occurred.

Program State Change

A program-state change does not accompany interrupts generated by the `INT n` instruction or the INTR signal. The `INT n` instruction generates the interrupt within the instruction stream. When the processor receives an INTR signal, it commits all state changes for all previous instructions before it responds to the interrupt; so, program execution can resume upon returning from the interrupt handler.

17. Updates to Chapter 8, Volume 3A

Change bars and green text show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Added details on protection keys for supervisor-mode pages where applicable. Typo corrections as needed.

The Intel 64 and IA-32 architectures provide mechanisms for managing and improving the performance of multiple processors connected to the same system bus. These include:

- Bus locking and/or cache coherency management for performing atomic operations on system memory.
- Serializing instructions.
- An advance programmable interrupt controller (APIC) located on the processor chip (see Chapter 10, “Advanced Programmable Interrupt Controller (APIC)”). This feature was introduced by the Pentium processor.
- A second-level cache (level 2, L2). For the Pentium 4, Intel Xeon, and P6 family processors, the L2 cache is included in the processor package and is tightly coupled to the processor. For the Pentium and Intel486 processors, pins are provided to support an external L2 cache.
- A third-level cache (level 3, L3). For Intel Xeon processors, the L3 cache is included in the processor package and is tightly coupled to the processor.
- Intel Hyper-Threading Technology. This extension to the Intel 64 and IA-32 architectures enables a single processor core to execute two or more threads concurrently (see Section 8.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology”).

These mechanisms are particularly useful in symmetric-multiprocessing (SMP) systems. However, they can also be used when an Intel 64 or IA-32 processor and a special-purpose processor (such as a communications, graphics, or video processor) share the system bus.

These multiprocessing mechanisms have the following characteristics:

- To maintain system memory coherency — When two or more processors are attempting simultaneously to access the same address in system memory, some communication mechanism or memory access protocol must be available to promote data coherency and, in some instances, to allow one processor to temporarily lock a memory location.
- To maintain cache consistency — When one processor accesses data cached on another processor, it must not receive incorrect data. If it modifies data, all other processors that access that data must receive the modified data.
- To allow predictable ordering of writes to memory — In some circumstances, it is important that memory writes be observed externally in precisely the same order as programmed.
- To distribute interrupt handling among a group of processors — When several processors are operating in a system in parallel, it is useful to have a centralized mechanism for receiving interrupts and distributing them to available processors for servicing.
- To increase system performance by exploiting the multi-threaded and multi-process nature of contemporary operating systems and applications.

The caching mechanism and cache consistency of Intel 64 and IA-32 processors are discussed in Chapter 11. The APIC architecture is described in Chapter 10. Bus and memory locking, serializing instructions, memory ordering, and Intel Hyper-Threading Technology are discussed in the following sections.

8.1 LOCKED ATOMIC OPERATIONS

The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- Guaranteed atomic operations
- Bus locking, using the LOCK# signal and the LOCK instruction prefix

- Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols ensure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

NOTE

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

The mechanisms for handling locked atomic operations have evolved with the complexity of IA-32 processors. More recent IA-32 processors (such as the Pentium 4, Intel Xeon, and P6 family processors) and Intel 64 provide a more refined locking mechanism than earlier processors. These mechanisms are described in the following sections.

8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel[®] Atom™, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.

An x87 instruction or an SSE instructions that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory, some of the accesses may complete (writing to memory) while another causes the operation to fault for architectural reasons (e.g. due an page-table entry that is marked "not present"). In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault. If TLB invalidation has been delayed (see Section 4.10.4.4), such page faults may occur even if all accesses are to the same page.

8.1.2 Bus Locking

Intel 64 and IA-32 processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked. Software can specify other occasions when the LOCK semantics are to be followed by prepending the LOCK prefix to an instruction.

In the case of the Intel386, Intel486, and Pentium processors, explicitly locked instructions will result in the assertion of the LOCK# signal. It is the responsibility of the hardware designer to make the LOCK# signal available in system hardware to control memory accesses among processors.

For the P6 and more recent processor families, if the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted; instead, locking is only applied to the processor's caches (see Section 8.1.4, "Effects of a LOCK Operation on Internal Processor Caches").

8.1.2.1 Automatic Locking

The operations on which the processor automatically follows the LOCK semantics are as follows:

- When executing an XCHG instruction that references memory.
- **When setting the B (busy) flag of a TSS descriptor** — The processor tests and sets the busy flag in the type field of the TSS descriptor when switching to a task. To ensure that two processors do not switch to the same task simultaneously, the processor follows the LOCK semantics while testing and setting this flag.
- **When updating segment descriptors** — When loading a segment descriptor, the processor will set the accessed flag in the segment descriptor if the flag is clear. During this operation, the processor follows the LOCK semantics so that the descriptor will not be modified by another processor while it is being updated. For this action to be effective, operating-system procedures that update descriptors should use the following steps:
 - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is not-present, and specify a value for the type field that indicates that the descriptor is being updated.
 - Update the fields of the segment descriptor. (This operation may require several memory accesses; therefore, locked operations cannot be used.)
 - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is valid and present.
- The Intel386 processor always updates the accessed flag in the segment descriptor, whether it is clear or not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors only update this flag if it is not already set.
- **When updating page-directory and page-table entries** — When updating page-directory and page-table entries, the processor uses locked cycles to set the accessed and dirty flag in the page-directory and page-table entries.
- **Acknowledging interrupts** — After an interrupt request, an interrupt controller may use the data bus to send the interrupt's vector to the processor. The processor follows the LOCK semantics during this time to ensure that no other data appears on the data bus while the vector is being transmitted.

8.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, and CMPXCHG8B).
- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signalling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access.

NOTE

Do not implement semaphores using the WC memory type. Do not perform non-temporal stores to a cache line containing a location used to implement a semaphore.

The integrity of a bus lock is not affected by the alignment of the memory field. The LOCK semantics are followed for as many bus cycles as necessary to update the entire operand. However, it is recommended that locked accesses be aligned on their natural boundaries for better system performance:

- Any boundary for an 8-bit access (locked or otherwise).
- 16-bit boundary for locked word accesses.
- 32-bit boundary for locked doubleword accesses.
- 64-bit boundary for locked quadword accesses.

Locked operations are atomic with respect to all other memory operations and all externally visible events. Only instruction fetch and page table accesses can pass locked instructions. Locked instructions can be used to synchronize data written by one processor and read by another processor.

For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete). This rule is also true for the Pentium 4 and Intel Xeon processors, with one exception. Load operations that reference weakly ordered memory types (such as the WC memory type) may not be serialized.

Locked instructions should not be used to ensure that data written can be fetched as instructions.

NOTE

The locked instructions for the current versions of the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors allow data written to be fetched as instructions. However, Intel recommends that developers who require the use of self-modifying code use a different synchronizing mechanism, described in the following sections.

8.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

(* OPTION 1 *)

Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)

Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;

The use of one of these options is not required for programs intended to run on the Pentium or Intel486 processors, but are recommended to ensure compatibility with the P6 and more recent processor families.

Self-modifying code will execute at a lower level of performance than non-self-modifying or normal code. The degree of the performance deterioration will depend upon the frequency of modification and specific characteristics of the code.

The act of one processor writing data into the currently executing code segment of a second processor with the intent of having the second processor execute that data as code is called **cross-modifying code**. As with self-modifying code, IA-32 processors exhibit model-specific behavior when executing cross-modifying code, depending upon how far ahead of the executing processors current execution pointer the code has been modified.

To write cross-modifying code and ensure that it is compliant with current and future versions of the IA-32 architecture, the following processor synchronization algorithm must be implemented:

```
(* Action of Modifying Processor *)
Memory_Flag := 0; (* Set Memory_Flag to value other than 1 *)
Store modified code (as data) into code segment;
Memory_Flag := 1;

(* Action of Executing Processor *)
WHILE (Memory_Flag ≠ 1)
    Wait for code to update;
ELIHW;
Execute serializing instruction; (* For example, CPUID instruction *)
Begin executing modified code;
```

(The use of this option is not required for programs intended to run on the Intel486 processor, but is recommended to ensure compatibility with the Pentium 4, Intel Xeon, P6 family, and Pentium processors.)

Like self-modifying code, cross-modifying code will execute at a lower level of performance than non-cross-modifying (normal) code, depending upon the frequency of modification and specific characteristics of the code.

The restrictions on self-modifying code and cross-modifying code also apply to the Intel 64 architecture.

8.1.4 Effects of a LOCK Operation on Internal Processor Caches

For the Intel486 and Pentium processors, the LOCK# signal is always asserted on the bus during a LOCK operation, even if the area of memory being locked is cached in the processor.

For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow its cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.

8.2 MEMORY ORDERING

The term **memory ordering** refers to the order in which the processor issues reads (loads) and writes (stores) through the system bus to system memory. The Intel 64 and IA-32 architectures support several memory-ordering models depending on the implementation of the architecture. For example, the Intel386 processor enforces **program ordering** (generally referred to as **strong ordering**), where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

To allow performance optimization of instruction execution, the IA-32 architecture allows departures from strong-ordering model called **processor ordering** in Pentium 4, Intel Xeon, and P6 family processors. These **processor-ordering** variations (called here the **memory-ordering model**) allow performance enhancing operations such as allowing reads to go ahead of buffered writes. The goal of any of these variations is to increase instruction execution speeds, while maintaining memory coherency, even in multiple-processor systems.

Section 8.2.1 and Section 8.2.2 describe the memory-ordering implemented by Intel486, Pentium, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors. Section 8.2.3 gives examples

illustrating the behavior of the memory-ordering model on IA-32 and Intel-64 processors. Section 8.2.4 considers the special treatment of stores for string operations and Section 8.2.5 discusses how memory-ordering behavior may be modified through the use of specific instructions.

8.2.1 Memory Ordering in the Intel® Pentium® and Intel486™ Processors

The Pentium and Intel486 processors follow the processor-ordered memory model; however, they operate as strongly-ordered processors under most circumstances. Reads and writes always appear in programmed order at the system bus—except for the following situation where processor ordering is exhibited. Read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cache hits and, therefore, are not directed to the same address being accessed by the read miss.

In the case of I/O operations, both reads and writes always appear in programmed order.

Software intended to operate correctly in processor-ordered processors (such as the Pentium 4, Intel Xeon, and P6 family processors) should not depend on the relatively strong ordering of the Pentium or Intel486 processors. Instead, it should ensure that accesses to shared variables that are intended to control concurrent execution among processors are explicitly required to obey program ordering through the use of appropriate locking or serializing operations (see Section 8.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term “processor” refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
 - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
 - string operations (see Section 8.2.4.1).
- No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written.¹ Executions of the CLFLUSH instruction are not reordered with each other. Executions of CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.
- MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.

1. Earlier versions of this manual specified that writes to memory may be reordered with executions of the CLFLUSH instruction. No processors implementing the CLFLUSH instruction allow such reordering.

In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).
- Any two stores are seen in a consistent order by processors other than those performing the stores
- Locked instructions have a total order.

See the example in Figure 8-1. Consider three processors in a system and each processor performs three writes, one to each of three defined locations (A, B, and C). Individually, the processors perform the writes in the same program order, but because of bus arbitration and other memory access mechanisms, the order that the three processors write the individual memory locations can differ each time the respective code sequences are executed on the processors. The final values in location A, B, and C would possibly vary on each execution of the write sequence.

The processor-ordering model described in this section is virtually identical to that used by the Pentium and Intel486 processors. The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are:

- Added support for speculative reads, while still adhering to the ordering principles above.
- Store-buffer forwarding, when a read passes a write to the same memory location.
- Out of order store from long string store and string move operations (see Section 8.2.4, "Fast-String Operation and Out-of-Order Stores," below).

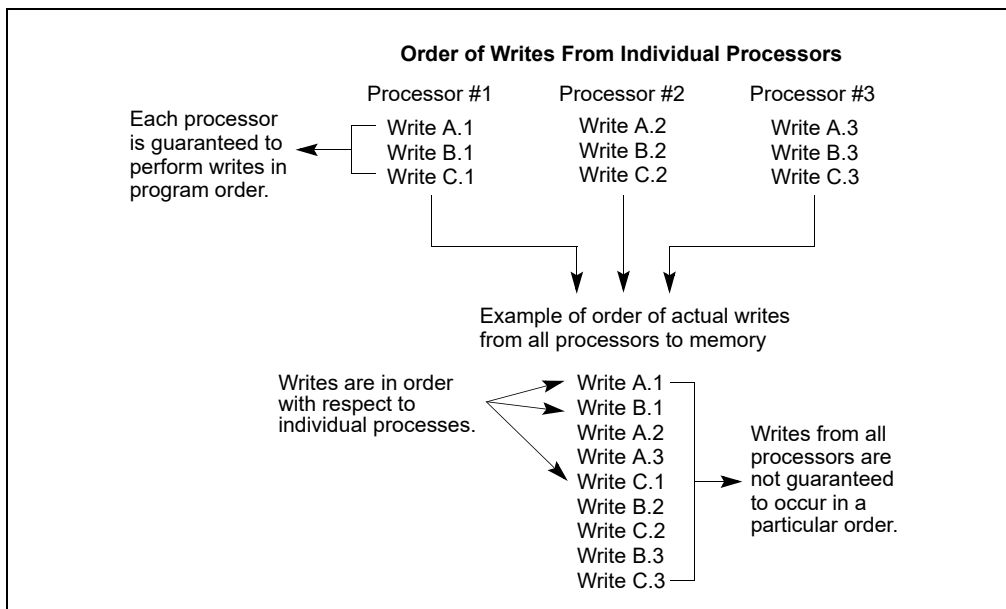


Figure 8-1. Example of Write Ordering in Multiple-Processor Systems

NOTE

In P6 processor family, store-buffer forwarding to reads of WC memory from streaming stores to the same address does not occur due to errata.

8.2.3 Examples Illustrating the Memory-Ordering Principles

This section provides a set of examples that illustrate the behavior of the memory-ordering principles introduced in Section 8.2.2. They are designed to give software writers an understanding of how memory ordering may affect the results of different sequences of instructions.

These examples are limited to accesses to memory regions defined as write-back cacheable (WB). (Section 8.2.3.1 describes other limitations on the generality of the examples.) The reader should understand that they describe only software-visible behavior. A logical processor may reorder two accesses even if one of examples indicates that they may not be reordered. Such an example states only that software cannot detect that such a reordering occurred. Similarly, a logical processor may execute a memory access more than once as long as the behavior visible to software is consistent with a single execution of the memory access.

8.2.3.1 Assumptions, Terminology, and Notation

As noted above, the examples in this section are limited to accesses to memory regions defined as write-back cacheable (WB). They apply only to ordinary loads stores and to locked read-modify-write instructions. They do not necessarily apply to any of the following: out-of-order stores for string instructions (see Section 8.2.4); accesses with a non-temporal hint; reads from memory by the processor as part of address translation (e.g., page walks); and updates to segmentation and paging structures by the processor (e.g., to update “accessed” bits).

The principles underlying the examples in this section apply to individual memory accesses and to locked read-modify-write instructions. The Intel-64 memory-ordering model guarantees that, for each of the following memory-access instructions, the constituent memory operation appears to execute as a single memory access:

- Instructions that read or write a single byte.
- Instructions that read or write a word (2 bytes) whose address is aligned on a 2 byte boundary.
- Instructions that read or write a doubleword (4 bytes) whose address is aligned on a 4 byte boundary.
- Instructions that read or write a quadword (8 bytes) whose address is aligned on an 8 byte boundary.

Any locked instruction (either the XCHG instruction or another read-modify-write instruction with a LOCK prefix) appears to execute as an indivisible and uninterruptible sequence of load(s) followed by store(s) regardless of alignment.

Other instructions may be implemented with multiple memory accesses. From a memory-ordering point of view, there are no guarantees regarding the relative order in which the constituent memory accesses are made. There is also no guarantee that the constituent operations of a store are executed in the same order as the constituent operations of a load.

Section 8.2.3.2 through Section 8.2.3.7 give examples using the MOV instruction. The principles that underlie these examples apply to load and store accesses in general and to other instructions that load from or store to memory. Section 8.2.3.8 and Section 8.2.3.9 give examples using the XCHG instruction. The principles that underlie these examples apply to other locked read-modify-write instructions.

This section uses the term “processor” is to refer to a logical processor. The examples are written using Intel-64 assembly-language syntax and use the following notational conventions:

- Arguments beginning with an “r”, such as r1 or r2 refer to registers (e.g., EAX) visible only to the processor being considered.
- Memory locations are denoted with x, y, z.
- Stores are written as *mov [_x], val*, which implies that *val* is being stored into the memory location *x*.
- Loads are written as *mov r, [_x]*, which implies that the contents of the memory location *x* are being loaded into the register *r*.

As noted earlier, the examples refer only to software visible behavior. When the succeeding sections make statement such as “the two stores are reordered,” the implication is only that “the two stores appear to be reordered from the point of view of software.”

8.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory-ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

Example 8-1. Stores Are Not Reordered with Other Stores

Processor 0	Processor 1
mov [_x], 1 mov [_y], 1	mov r1, [_y] mov r2, [_x]
Initially x = y = 0 r1 = 1 and r2 = 0 is not allowed	

The disallowed return values could be exhibited only if processor 0's two stores are reordered (with the two loads occurring between them) or if processor 1's two loads are reordered (with the two stores occurring between them).

If r1 = 1, the store to y occurs before the load from y. Because the Intel-64 memory-ordering model does not allow stores to be reordered, the earlier store to x occurs before the load from y. Because the Intel-64 memory-ordering model does not allow loads to be reordered, the store to x also occurs before the later load from x. This r2 = 1.

8.2.3.3 Stores Are Not Reordered with Earlier Loads

The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor. This is illustrated by the following example:

Example 8-2. Stores Are Not Reordered with Older Loads

Processor 0	Processor 1
mov r1, [_x] mov [_y], 1	mov r2, [_y] mov [_x], 1
Initially x = y = 0 r1 = 1 and r2 = 1 is not allowed	

Assume r1 = 1.

- Because r1 = 1, processor 1's store to x occurs before processor 0's load from x.
- Because the Intel-64 memory-ordering model prevents each store from being reordered with the earlier load by the same processor, processor 1's load from y occurs before its store to x.
- Similarly, processor 0's load from x occurs before its store to y.
- Thus, processor 1's load from y occurs before processor 0's store to y, implying r2 = 0.

8.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

Example 8-3. Loads May be Reordered with Older Stores

Processor 0	Processor 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]
Initially x = y = 0 r1 = 0 and r2 = 0 is allowed	

At each processor, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

Example 8-4. Loads Are not Reordered with Older Stores to the Same Location

Processor 0
mov [_x], 1 mov r1, [_x]
Initially x = 0 r1 = 0 is not allowed

The Intel-64 memory-ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, r1 = 1 must hold.

8.2.3.5 Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

Example 8-5. Intra-Processor Forwarding is Allowed

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

8.2.3.6 Stores Are Transitively Visible

The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation. This is illustrated by the following example:

Example 8-6. Stores Are Transitively Visible

Processor 0	Processor 1	Processor 2
mov [_x], 1	mov r1, [_x] mov [_y], 1	mov r2, [_y] mov r3, [_x]
Initially x = y = 0 r1 = 1, r2 = 1, r3 = 0 is not allowed		

Assume that r1 = 1 and r2 = 1.

- Because r1 = 1, processor 0's store occurs before processor 1's load.

- Because the memory-ordering model prevents a store from being reordered with an earlier load (see Section 8.2.3.3), processor 1’s load occurs before its store. Thus, processor 0’s store causally precedes processor 1’s store.
- Because processor 0’s store causally precedes processor 1’s store, the memory-ordering model ensures that processor 0’s store appears to occur before processor 1’s store from the point of view of all processors.
- Because $r2 = 1$, processor 1’s store occurs before processor 2’s load.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 2’s load occur in order.
- The above items imply that processor 0’s store to x occurs before processor 2’s load from x . This implies that $r3 = 1$.

8.2.3.7 Stores Are Seen in a Consistent Order by Other Processors

As noted in Section 8.2.3.5, the memory-ordering model allows stores by two processors to be seen in different orders by those two processors. However, any two stores must appear to execute in the same order to all processors other than those performing the stores. This is illustrated by the following example:

Example 8-7. Stores Are Seen in a Consistent Order by Other Processors

Processor 0	Processor 1	Processor 2	Processor 3
mov [_x], 1	mov [_y], 1	mov r1, [_x] mov r2, [_y]	mov r3, [_y] mov r4, [_x]
Initially $x = y = 0$ $r1 = 1, r2 = 0, r3 = 1, r4 = 0$ is not allowed			

By the principles discussed in Section 8.2.3.2,

- processor 2’s first and second load cannot be reordered,
- processor 3’s first and second load cannot be reordered.
- If $r1 = 1$ and $r2 = 0$, processor 0’s store appears to precede processor 1’s store with respect to processor 2.
- Similarly, $r3 = 1$ and $r4 = 0$ imply that processor 1’s store appears to precede processor 0’s store with respect to processor 1.

Because the memory-ordering model ensures that any two stores appear to execute in the same order to all processors (other than those performing the stores), this set of return values is not allowed

8.2.3.8 Locked Instructions Have a Total Order

The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions, including those that are larger than 8 bytes or are not naturally aligned. This is illustrated by the following example:

Example 8-8. Locked Instructions Have a Total Order

Processor 0	Processor 1	Processor 2	Processor 3
xchg [_x], r1	xchg [_y], r2	mov r3, [_x] mov r4, [_y]	mov r5, [_y] mov r6, [_x]
Initially $r1 = r2 = 1, x = y = 0$ $r3 = 1, r4 = 0, r5 = 1, r6 = 0$ is not allowed			

Processor 2 and processor 3 must agree on the order of the two executions of XCHG. Without loss of generality, suppose that processor 0’s XCHG occurs first.

- If $r5 = 1$, processor 1’s XCHG into y occurs before processor 3’s load from y .

- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 3’s loads occur in order and, therefore, processor 1’s XCHG occurs before processor 3’s load from x.
- Since processor 0’s XCHG into x occurs before processor 1’s XCHG (by assumption), it occurs before processor 3’s load from x. Thus, $r6 = 1$.

A similar argument (referring instead to processor 2’s loads) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

8.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

Example 8-9. Loads Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov r2, [_y]	xchg [_y], r3 mov r4, [_x]
Initially $x = y = 0, r1 = r3 = 1$ $r2 = 0$ and $r4 = 0$ is not allowed	

As explained in Section 8.2.3.8, there is a total order of the executions of locked instructions. Without loss of generality, suppose that processor 0’s XCHG occurs first.

Because the Intel-64 memory-ordering model prevents processor 1’s load from being reordered with its earlier XCHG, processor 0’s XCHG occurs before processor 1’s load. This implies $r4 = 1$.

A similar argument (referring instead to processor 2’s accesses) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

Example 8-10. Stores Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov [_y], 1	mov r2, [_y] mov r3, [_x]
Initially $x = y = 0, r1 = 1$ $r2 = 1$ and $r3 = 0$ is not allowed	

Assume $r2 = 1$.

- Because $r2 = 1$, processor 0’s store to y occurs before processor 1’s load from y.
- Because the memory-ordering model prevents a store from being reordered with an earlier locked instruction, processor 0’s XCHG into x occurs before its store to y. Thus, processor 0’s XCHG into x occurs before processor 1’s load from y.
- Because the memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 1’s loads occur in order and, therefore, processor 1’s XCHG into x occurs before processor 1’s load from x. Thus, $r3 = 1$.

8.2.4 Fast-String Operation and Out-of-Order Stores

Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* described an optimization of repeated string operations called **fast-string operation**.

As explained in that section, the stores produced by fast-string operation may appear to execute out of order. Software dependent upon sequential store ordering should not use string operations for the entire data structure to be stored. Data and semaphores should be separated. Order-dependent code should write to a discrete semaphore variable after any string operations to allow correctly ordered data to be seen by all processors. Atomicity of load and store operations is guaranteed only for native data elements of the string with native data size, and only if they are included in a single cache line.

Section 8.2.4.1 and Section 8.2.4.2 provide further explain and examples.

8.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory

This section deals with the memory-ordering model for string operations on write-back (WB) memory for the Intel 64 architecture.

The memory-ordering model respects the follow principles:

1. Stores within a single string operation may be executed out of order.
2. Stores from separate string operations (for example, stores from consecutive string operations) do not execute out of order. All the stores from an earlier string operation will complete before any store from a later string operation.
3. String operations are not reordered with other store operations.

Fast string operations (e.g. string operations initiated with the MOVSB/STOSB instructions and the REP prefix) may be interrupted by exceptions or interrupts. The interrupts are precise but may be delayed - for example, the interruptions may be taken at cache line boundaries, after every few iterations of the loop, or after operating on every few bytes. Different implementations may choose different options, or may even choose not to delay interrupt handling, so software should not rely on the delay. When the interrupt/trap handler is reached, the source/destination registers point to the next string element to be operated on, while the EIP stored in the stack points to the string instruction, and the ECX register has the value it held following the last successful iteration. The return from that trap/interrupt handler should cause the string instruction to be resumed from the point where it was interrupted.

The string operation memory-ordering principles, (item 2 and 3 above) should be interpreted by taking the incorruptibility of fast string operations into account. For example, if a fast string operation gets interrupted after k iterations, then stores performed by the interrupt handler will become visible after the fast string stores from iteration 0 to k, and before the fast string stores from the (k+1)th iteration onward.

Stores within a single string operation may execute out of order (item 1 above) only if fast string operation is enabled. Fast string operations are enabled/disabled through the IA32_MISC_ENABLE model specific register.

8.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations

The following examples uses the same notation and convention as described in Section 8.2.3.1.

In Example 8-11, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location `_x` (kept in ES:EDI) in ascending order. Since each operation stores a doubleword (4 bytes), the operation is repeated 128 times (value in ECX). The block of memory initially contained 0. Processor 1 is reading two memory locations that are part of the memory block being updated by processor 0, i.e, reading locations in the range `_x` to `(_x+511)`.

Example 8-11. Stores Within a String Operation May be Reordered

Processor 0	Processor 1
<code>rep:stosd [_x]</code>	<code>mov r1, [_z]</code> <code>mov r2, [_y]</code>
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = <code>_x</code> Initially <code>[_x]</code> to <code>511[_x]</code> = 0, <code>_x <= _y < _z < _x+512</code> <code>r1 = 1</code> and <code>r2 = 0</code> is allowed	

MULTIPLE-PROCESSOR MANAGEMENT

It is possible for processor 1 to perceive that the repeated string stores in processor 0 are happening out of order. Assume that fast string operations are enabled on processor 0.

In Example 8-12, processor 0 does two separate rounds of `rep stosd` operation of 128 doubleword stores, writing the value 1 (value in `EAX`) into the first block of 512 bytes from location `_x` (kept in `ES:EDI`) in ascending order. It then writes 1 into a second block of memory from `(_x+512)` to `(_x+1023)`. All of the memory locations initially contain 0. The block of memory initially contained 0. Processor 1 performs two load operations from the two blocks of memory.

Example 8-12. Stores Across String Operations Are not Reordered

Processor 0	Processor 1
<code>rep:stosd [_x]</code>	<code>mov r1, [_z]</code>
<code>mov ecx, \$128</code>	<code>mov r2, [_y]</code>
<code>rep:stosd 512[_x]</code>	
Initially on processor 0: <code>EAX = 1, ECX=128, ES:EDI =_x</code> Initially <code>[_x] to 1023[_x]= 0, _x <= _y < _x+512 < _z < _x+1024</code> <code>r1 = 1 and r2 = 0</code> is not allowed	

It is not possible in the above example for processor 1 to perceive any of the stores from the later string operation (to the second 512 block) in processor 0 before seeing the stores from the earlier string operation to the first 512 block.

The above example assumes that writes to the second block (`_x+512` to `_x+1023`) does not get executed while processor 0's string operation to the first block has been interrupted. If the string operation to the first block by processor 0 is interrupted, and a write to the second memory block is executed by the interrupt handler, then that change in the second memory block will be visible before the string operation to the first memory block resumes.

In Example 8-13, processor 0 does one round of (128 iterations) doubleword string store operation via `rep:stosd`, writing the value 1 (value in `EAX`) into a block of 512 bytes from location `_x` (kept in `ES:EDI`) in ascending order. It then writes to a second memory location outside the memory block of the previous string operation. Processor 1 performs two read operations, the first read is from an address outside the 512-byte block but to be updated by processor 0, the second read is from inside the block of memory of string operation.

Example 8-13. String Operations Are not Reordered with later Stores

Processor 0	Processor 1
<code>rep:stosd [_x]</code>	<code>mov r1, [_z]</code>
<code>mov [_z], \$1</code>	<code>mov r2, [_y]</code>
Initially on processor 0: <code>EAX = 1, ECX=128, ES:EDI =_x</code> Initially <code>[_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z</code> is a separate memory location <code>r1 = 1 and r2 = 0</code> is not allowed	

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example 8-13 assumes that processor 0's store to `[_z]` is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to `[_z]` by processor 0 is executed by the interrupt handler, then changes to `[_z]` will become visible before the string operation resumes.

Example 8-14 illustrates the visibility principle when a string operation is interrupted.

Example 8-14. Interrupted String Operation

Processor 0	Processor 1
rep:stosd [_x] // interrupted before es:edi reach _y mov [_z], \$1 // interrupt handler	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is allowed	

In Example 8-14, processor 0 started a string operation to write to a memory block of 512 bytes starting at address _x. Processor 0 got interrupted after k iterations of store operations. The address _y has not yet been updated by processor 0 when processor 0 got interrupted. The interrupt handler that took control on processor 0 writes to the address _z. Processor 1 may see the store to _z from the interrupt handler, before seeing the remaining stores to the 512-byte memory block that are executed when the string operation resumes.

Example 8-15 illustrates the ordering of string operations with earlier stores. No store from a string operation can be visible before all prior stores are visible.

Example 8-15. String Operations Are not Reordered with Earlier Stores

Processor 0	Processor 1
mov [_z], \$1 rep:stosd [_x]	mov r1, [_y] mov r2, [_z]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

8.2.5 Strengthening or Weakening the Memory-Ordering Model

The Intel 64 and IA-32 architectures provide several mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations. These mechanisms include:

- The I/O instructions, locking instructions, the LOCK prefix, and serializing instructions force stronger ordering on the processor.
- The SFENCE instruction (introduced to the IA-32 architecture in the Pentium III processor) and the LFENCE and MFENCE instructions (introduced in the Pentium 4 processor) provide memory-ordering and serialization capabilities for specific types of memory operations.
- The memory type range registers (MTRRs) can be used to strengthen or weaken memory ordering for specific area of physical memory (see Section 11.11, "Memory Type Range Registers (MTRRs)"). MTRRs are available only in the Pentium 4, Intel Xeon, and P6 family processors.
- The page attribute table (PAT) can be used to strengthen memory ordering for a specific page or group of pages (see Section 11.12, "Page Attribute Table (PAT)"). The PAT is available only in the Pentium 4, Intel Xeon, and Pentium III processors.

These mechanisms can be used as follows:

Memory mapped devices and other I/O devices on the bus are often sensitive to the order of writes to their I/O buffers. I/O instructions can be used to (the IN and OUT instructions) impose strong write ordering on such accesses as follows. Prior to executing an I/O instruction, the processor waits for all previous instructions in the program to complete and for all buffered writes to drain to memory. Only instruction fetch and page tables walks can pass I/O instructions. Execution of subsequent instructions do not begin until the processor determines that the I/O instruction has been completed.

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locking instruction such as the XCHG instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out atomically. Locking operations typically operate like I/O operations in that they wait for all previous instructions to complete and for all buffered writes to drain to memory (see Section 8.1.2, “Bus Locking”).

Program synchronization can also be carried out with serializing instructions (see Section 8.3). These instructions are typically used at critical procedure or task boundaries to force completion of all previous instructions before a jump to a new section of code or a context switch occurs. Like the I/O and locking instructions, the processor waits until all previous instructions have been completed and all buffered writes have been drained to memory before executing the serializing instruction.

The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.²
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Note that the SFENCE, LFENCE, and MFENCE instructions provide a more efficient method of controlling memory ordering than the CPUID instruction.

The MTRRs were introduced in the P6 family processors to define the cache characteristics for specified areas of physical memory. The following are two examples of how memory types set up with MTRRs can be used strengthen or weaken memory ordering for the Pentium 4, Intel Xeon, and P6 family processors:

- The strong uncached (UC) memory type forces a strong-ordering model on memory accesses. Here, all reads and writes to the UC memory region appear on the bus and out-of-order or speculative accesses are not performed. This memory type can be applied to an address range dedicated to memory mapped I/O devices to force strong memory ordering.
- For areas of memory where weak ordering is acceptable, the write back (WB) memory type can be chosen. Here, reads can be performed speculatively and writes can be buffered and combined. For this type of memory, cache locking is performed on atomic (locked) operations that do not split across cache lines, which helps to reduce the performance penalty associated with the use of the typical synchronization instructions, such as XCHG, that lock the bus during the entire read-modify-write operation. With the WB memory type, the XCHG instruction locks the cache instead of the bus if the memory access is contained within a cache line.

The PAT was introduced in the Pentium III processor to enhance the caching characteristics that can be assigned to pages or groups of pages. The PAT mechanism typically used to strengthen caching characteristics at the page level with respect to the caching characteristics established by the MTRRs. Table 11-7 shows the interaction of the PAT with the MTRRs.

Intel recommends that software written to run on Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors assume the processor-ordering model or a weaker memory-ordering model. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors do not implement a strong memory-ordering model, except when using the UC memory type. Despite the fact that Pentium 4, Intel Xeon, and P6 family processors support processor ordering, Intel does not guarantee that future processors will support this model. To make software portable to future processors, it is recommended that operating systems provide critical region and resource control constructs and API's (application program interfaces) based on I/O, locking, and/or serializing instructions be used to synchronize access to shared areas of memory in multiple-processor systems. Also, software should not depend on processor ordering in situations where the system hardware does not support this memory-ordering model.

-
2. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. As a result, 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.

8.3 SERIALIZING INSTRUCTIONS

The Intel 64 and IA-32 architectures define several **serializing instructions**. These instructions force the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. For example, when a MOV to control register instruction is used to load a new value into control register CR0 to enable protected mode, the processor must perform a serializing operation before it enters protected mode. This serializing operation ensures that all operations that were started while the processor was in real-address mode are completed before the switch to protected mode is made.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution.

It is important to note that executing of serializing instructions on P6 and more recent processor families constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — INVD, INVEPT, INVLPG, INVVPID, LGDT, LIDT, LLDT, LTR, MOV (to control register, with the exception of MOV CR8³), MOV (to debug register), WBINVD, and WRMSR⁴.
- **Non-privileged serializing instructions** — CPUID, IRET, and RSM.

When the processor serializes instruction execution, it ensures that all pending memory transactions are completed (including writes stored in its store buffer) before it executes the next instruction. Nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction (read, write, instruction fetch, or I/O). For example, CPUID can be executed at any privilege level to serialize instruction execution with no effect on program flow, except that the EAX, EBX, ECX, and EDX registers are modified.

The following instructions are memory-ordering instructions, not serializing instructions. These drain the data memory subsystem. They do not serialize the instruction execution stream:⁵

- **Non-privileged memory-ordering instructions** — SFENCE, LFENCE, and MFENCE.

The SFENCE, LFENCE, and MFENCE instructions provide more granularity in controlling the serialization of memory loads and stores (see Section 8.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

The following additional information is worth noting regarding serializing instructions:

- The processor does not write back the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction, which is a serializing instruction. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- When an instruction is executed that enables or disables paging (that is, changes the PG flag in control register CR0), the instruction should be followed by a jump instruction. The target instruction of the jump instruction is fetched with the new setting of the PG flag (that is, paging is enabled or disabled), but the jump instruction itself is fetched with the previous setting. The Pentium 4, Intel Xeon, and P6 family processors do not require the jump operation following the move to register CR0 (because any use of the MOV instruction in a Pentium 4, Intel Xeon, or P6 family processor to write to CR0 is completely serializing). However, to maintain backwards and forward compatibility with code written to run on other IA-32 processors, it is recommended that the jump operation be performed.

3. MOV CR8 is not defined architecturally as a serializing instruction.

4. An execution of WRMSR to any non-serializing MSR is not serializing. Non-serializing MSRs include the following: IA32_SPEC_CTRL MSR (MSR index 48H), IA32_PRED_CMD MSR (MSR index 49H), IA32_TSX_CTRL MSR (MSR index 122H), IA32_TSC_DEADLINE MSR (MSR index 6E0H), IA32_PKRS MSR (MSR index 6E1H), IA32_HWP_REQUEST MSR (MSR index 774H), or any of the x2APIC MSRs (MSR indices 802H to 83FH).

5. LFENCE does provide some guarantees on instruction ordering. It does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes.

- Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3. (Global entries in the TLBs are not invalidated, see Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches.”)
- The Pentium processor and more recent processor families use branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not deterministically serialized when a branch instruction is executed.

8.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION

The IA-32 architecture (beginning with the P6 family processors) defines a multiple-processor (MP) initialization protocol called the *Multiprocessor Specification Version 1.4*. This specification defines the boot protocol to be used by IA-32 processors in multiple-processor systems. (Here, **multiple processors** is defined as two or more processors.) The MP initialization protocol has the following important features:

- It supports controlled booting of multiple processors without requiring dedicated system hardware.
- It allows hardware to initiate the booting of a system without the need for a dedicated signal or a predefined boot processor.
- It allows all IA-32 processors to be booted in the same manner, including those supporting Intel Hyper-Threading Technology.
- The MP initialization protocol also applies to MP systems using Intel 64 processors.

The mechanism for carrying out the MP initialization protocol differs depending on the Intel processor generations. The following bullets summarize the evolution of the changes:

- **For P6 family or older processors supporting MP operations**— The selection of the BSP and APs (see Section 8.4.1, “BSP and AP Processors”) is handled through arbitration on the APIC bus, using BIPI and FIPI messages. These processor generations have CPUID signatures of (family=06H, extended_model=0, model<=0DH), or family <06H. See Section 8.11.1, “Overview of the MP Initialization Process For P6 Family Processors” for a complete discussion of MP initialization for P6 family processors.
- **Early generations of IA processors with family 0FH** — The selection of the BSP and APs (see Section 8.4.1, “BSP and AP Processors”) is handled through arbitration on the system bus, using BIPI and FIPI messages (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH, model=0H, stepping<=09H.
- **Later generations of IA processors with family 0FH, and IA processors with system bus** — The selection of the BSP and APs is handled through a special system bus cycle, without using BIPI and FIPI message arbitration (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH with (model=0H, stepping>=0AH) or (model >0, all steppings); or family=06H, extended_model=0, model>=0EH.
- **All other modern IA processor generations supporting MP operations**— The selection of the BSP and APs in the system is handled by platform-specific arrangement of the combination of hardware, BIOS, and/or configuration input options. The basis of the selection mechanism is similar to those of the Later generations of family 0FH and other Intel processor using system bus (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=06H, extended_model>0.

The family, model, and stepping ID for a processor is given in the EAX register when the CPUID instruction is executed with a value of 1 in the EAX register.

8.4.1 BSP and AP Processors

The MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs.

As part of the BSP selection mechanism, the BSP flag is set in the IA32_APIC_BASE MSR (see Figure 10-5) of the BSP, indicating that it is the BSP. This flag is cleared for all other processors.

The BSP executes the BIOS's boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs. When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code.

Following a power-up or reset, the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor. Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state.

For Intel 64 and IA-32 processors supporting Intel Hyper-Threading Technology, the MP initialization protocol treats each of the logical processors on the system bus or coherent link domain as a separate processor (with a unique APIC ID). During boot-up, one of the logical processors is selected as the BSP and the remainder of the logical processors are designated as APs.

8.4.2 MP Initialization Protocol Requirements and Restrictions

The MP initialization protocol imposes the following requirements and restrictions on the system:

- The MP protocol is executed only after a power-up or RESET. If the MP protocol has completed and a BSP is chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each logical processor examines its BSP flag (in the IA32_APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

8.4.3 MP Initialization Protocol Algorithm for MP Systems

Following a power-up or RESET of an MP system, the processors in the system execute the MP initialization protocol algorithm to initialize each of the logical processors on the system bus or coherent link domain. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each logical processor is assigned a unique APIC ID, based on system topology. The unique ID is a 32-bit value if the processor supports CPUID leaf 0BH, otherwise the unique ID is an 8-bit value. (see Section 8.4.5, "Identifying Logical Processors in an MP System").
2. Each logical processor is assigned a unique arbitration priority based on its APIC ID.
3. Each logical processor executes its internal BIST simultaneously with the other logical processors in the system.
4. Upon completion of the BIST, the logical processors use a hardware-defined selection mechanism to select the BSP and the APs from the available logical processors on the system bus. The BSP selection mechanism differs depending on the family, model, and stepping IDs of the processors, as follows:
 - Later generations of IA processors within family 0FH (see Section 8.4), IA processors with system bus (family=06H, extended_model=0, model>=0EH), or all other modern Intel processors (family=06H, extended_model>0):
 - The logical processors begin monitoring the BNR# signal, which is toggling. When the BNR# pin stops toggling, each processor attempts to issue a NOP special cycle on the system bus.
 - The logical processor with the highest arbitration priority succeeds in issuing a NOP special cycle and is nominated the BSP. This processor sets the BSP flag in its IA32_APIC_BASE MSR, then fetches and begins executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
 - The remaining logical processors (that failed in issuing a NOP special cycle) are designated as APs. They leave their BSP flags in the clear state and enter a "wait-for-SIPI state."

- Early generations of IA processors within family 0FH (family=0FH, model=0H, stepping<=09H), P6 family or older processors supporting MP operations (family=06H, extended_model=0, model<=0DH; or family <06H):
 - Each processor broadcasts a BIPI to “all including self.” The first processor that broadcasts a BIPI (and thus receives its own BIPI vector), selects itself as the BSP and sets the BSP flag in its IA32_APIC_BASE MSR. (See Section 8.11.1, “Overview of the MP Initialization Process For P6 Family Processors” for a description of the BIPI, FIPI, and SIPI messages.)
 - The remainder of the processors (which were not selected as the BSP) are designated as APs. They leave their BSP flags in the clear state and enter a “wait-for-SIPI state.”
 - The newly established BSP broadcasts an FIPI message to “all including self,” which the BSP and APs treat as an end of MP initialization signal. Only the processor with its BSP flag set responds to the FIPI message. It responds by fetching and executing the BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
- 5. As part of the boot-strap code, the BSP creates an ACPI table and/or an MP table and adds its initial APIC ID to these tables as appropriate.
- 6. At the end of the boot-strap procedure, the BSP sets a processor counter to 1, then broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000VV000H, where VV is the vector contained in the SIPI message).
- 7. The first action of the AP initialization code is to set up a race (among the APs) to a BIOS initialization semaphore. The first AP to the semaphore begins executing the initialization code. (See Section 8.4.4, “MP Initialization Example,” for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and/or MP tables as appropriate and increments the processor counter by 1. At the completion of the initialization procedure, the AP executes a CLI instruction and halts itself.
- 8. When each of the APs has gained access to the semaphore and executed the AP initialization code, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
- 9. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

The following section gives an example (with code) of the MP initialization protocol for of multiple processors operating in an MP configuration.

Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* describes how to program the LINT[0:1] pins of the processor’s local APICs after an MP configuration has been completed.

8.4.4 MP Initialization Example

The following example illustrates the use of the MP initialization protocol used to initialize processors in an MP system after the BSP and APs have been established. The code runs on Intel 64 or IA-32 processors that use a protocol. This includes P6 Family processors, Pentium 4 processors, Intel Core Duo, Intel Core 2 Duo and Intel Xeon processors.

The following constants and data definitions are used in the accompanying code examples. They are based on the addresses of the APIC registers defined in Table 10-1.

```

ICR_LOW      EQU 0FEE00300H
SVR          EQU 0FEE000F0H
APIC_ID      EQU 0FEE00020H
LVT3        EQU 0FEE00370H
APIC_ENABLED EQU 0100H
BOOT_ID      DD ?
COUNT       EQU 00H
VACANT       EQU 00H
    
```

8.4.4.1 Typical BSP Initialization Sequence

After the BSP and APs have been selected (by means of a hardware protocol, see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”), the BSP begins executing BIOS boot-strap code (POST) at the normal IA-32 architecture starting address (FFFF FFF0H). The boot-strap code typically performs the following operations:

1. Initializes memory.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs.
4. Enables the caches.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the BSP is “GenuineIntel.”
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Loads start-up code for the AP to execute into a 4-KByte page in the lower 1 MByte of memory.
8. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
9. Determine the BSP’s APIC ID from the local APIC ID register (default is 0), the code snippet below is an example that applies to logical processors in a system whose local APIC units operate in xAPIC mode that APIC registers are accessed using memory mapped interface:

```
MOV ESI, APIC_ID; Address of local APIC ID register
MOV EAX, [ESI];
AND EAX, 0FF000000H; Zero out all other bits except APIC ID
MOV BOOT_ID, EAX; Save in memory
```

Saves the APIC ID in the ACPI and/or MP tables and optionally in the system configuration space in RAM.

10. Converts the base address of the 4-KByte page for the AP’s bootup code into 8-bit vector. The 8-bit vector defines the address of a 4-KByte page in the real-address mode address space (1-MByte space). For example, a vector of 0BDH specifies a start-up memory address of 000BD000H.
11. Enables the local APIC by setting bit 8 of the APIC spurious vector register (SVR).


```
MOV ESI, SVR; Address of SVR
MOV EAX, [ESI];
OR EAX, APIC_ENABLED; Set bit 8 to enable (0 on reset)
MOV [ESI], EAX;
```
12. Sets up the LVT error handling entry by establishing an 8-bit vector for the APIC error handler.

```
MOV ESI, LVT3;
MOV EAX, [ESI];
AND EAX, 0FFFFFF0H; Clear out previous vector.
OR EAX, 000000xxH; xx is the 8-bit vector the APIC error handler.
MOV [ESI], EAX;
```

13. Initializes the Lock Semaphore variable VACANT to 00H. The APs use this semaphore to determine the order in which they execute BIOS AP initialization code.
14. Performs the following operation to set up the BSP to detect the presence of APs in the system and the number of processors (within a finite duration, minimally 100 milliseconds):
 - Sets the value of the COUNT variable to 1.
 - In the AP BIOS initialization code, the AP will increment the COUNT variable to indicate its presence. The finite duration while waiting for the COUNT to be updated can be accomplished with a timer. When the timer expires, the BSP checks the value of the COUNT variable. If the timer expires and the COUNT variable has not been incremented, no APs are present or some error has occurred.

15. Broadcasts an INIT-SIPI-SIPI IPI sequence to the APs to wake them up and initialize them. If software knows how many logical processors it expects to wake up, it may choose to poll the COUNT variable. If the expected processors show up before the 100 millisecond timer expires, the timer can be canceled and skip to step 16. The left-hand-side of the procedure illustrated in Table 8-1 provides an algorithm when the expected processor count is unknown. The right-hand-side of Table 8-1 can be used when the expected processor count is known.

Table 8-1. Broadcast INIT-SIPI-SIPI Sequence and Choice of Timeouts

INIT-SIPI-SIPI when the expected processor count is unknown	INIT-SIPI-SIPI when the expected processor count is known
MOV ESI, ICR_LOw; Load address of ICR low dword into ESI. MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI ; to all APs into EAX. MOV [ESI], EAX; Broadcast INIT IPI to all APs ; 10-millisecond delay loop. MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP ; to all APs into EAX, where xx is the vector computed in step 10. MOV [ESI], EAX; Broadcast SIPI IPI to all APs ; 200-microsecond delay loop MOV [ESI], EAX; Broadcast second SIPI IPI to all APs ; Waits for the timer interrupt until the timer expires	MOV ESI, ICR_LOw; Load address of ICR low dword into ESI. MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI ; to all APs into EAX. MOV [ESI], EAX; Broadcast INIT IPI to all APs ; 10-millisecond delay loop. MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP ; to all APs into EAX, where xx is the vector computed in step 10. MOV [ESI], EAX; Broadcast SIPI IPI to all APs ; 200 microsecond delay loop with check to see if COUNT has ; reached the expected processor count. If COUNT reaches ; expected processor count, cancel timer and go to step 16. MOV [ESI], EAX; Broadcast second SIPI IPI to all APs ; Wait for the timer interrupt polling COUNT. If COUNT reaches ; expected processor count, cancel timer and go to step 16. ; If timer expires, go to step 16.

16. Reads and evaluates the COUNT variable and establishes a processor count.

17. If necessary, reconfigures the APIC and continues with the remaining system diagnostics as appropriate.

8.4.4.2 Typical AP Initialization Sequence

When an AP receives the SIPI, it begins executing BIOS AP initialization code at the vector encoded in the SIPI. The AP initialization code typically performs the following operations:

1. Waits on the BIOS initialization Lock Semaphore. When control of the semaphore is attained, initialization continues.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs (using the same mapping that was used for the BSP).
4. Enables the cache.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the AP is "GenuineIntel."
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
8. Determines the AP's APIC ID from the local APIC ID register, and adds it to the MP and ACPI tables and optionally to the system configuration space in RAM.
9. Initializes and configures the local APIC by setting bit 8 in the SVR register and setting up the LVT3 (error LVT) for error handling (as described in steps 9 and 10 in Section 8.4.4.1, "Typical BSP Initialization Sequence").

10. Configures the APs SMI execution environment. (Each AP and the BSP must have a different SMBASE address.)
11. Increments the COUNT variable by 1.
12. Releases the semaphore.
13. Executes one of the following:
 - the CLI and HLT instructions (if MONITOR/MWAIT is not supported), or
 - the CLI, MONITOR and MWAIT sequence to enter a deep C-state.
14. Waits for an INIT IPI.

8.4.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can read APIC ID in one of two ways depending on the local APIC unit is operating in x2APIC mode (see *Intel® 64 Architecture x2APIC Specification*) or in xAPIC mode:
 - If the local APIC unit supports x2APIC and is operating in x2APIC mode, 32-bit APIC ID can be read by executing a RDMSR instruction to read the processor's x2APIC ID register. This method is equivalent to executing CPUID leaf 0BH described below.
 - If the local APIC unit is operating in xAPIC mode, 8-bit APIC ID can be read by executing a MOV instruction to read the processor's local APIC ID register (see Section 10.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** (If the processor does not support CPUID leaf 0BH) — An APIC ID is assigned to a logical processor during power up. This is the initial APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. The initial APIC ID can be used to determine the topological relationship between logical processors for multi-processor systems that do not support CPUID leaf 0BH.

Bits in the 8-bit initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical level of the multi-threading resource topology in an MP system (See Section 8.9.1, "Hierarchical Mapping of Shared Resources"). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.
- **Read 32-bit APIC ID from CPUID leaf 0BH** (If the processor supports CPUID leaf 0BH) — A unique APIC ID is assigned to a logical processor during power up. This APIC ID is reported by CPUID.0BH:EDX[31:0] as a 32-bit value. Use the 32-bit APIC ID and CPUID leaf 0BH to determine the topological relationship between logical processors if the processor supports CPUID leaf 0BH.

Bits in the 32-bit x2APIC ID can be extracted into sub-fields using CPUID leaf 0BH parameters. (See Section 8.9.1, "Hierarchical Mapping of Shared Resources").

Figure 8-2 shows two examples of APIC ID bit fields in earlier single-core processors. In single-core Intel Xeon processors, the APIC ID assigned to a logical processor during power-up and initialization is 8 bits. Bits 2:1 form a 2-bit physical package identifier (which can also be thought of as a socket identifier). In systems that configure physical processors in clusters, bits 4:3 form a 2-bit cluster ID. Bit 0 is used in the Intel Xeon processor MP to identify the two logical processors within the package (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System"). For Intel Xeon processors that do not support Intel Hyper-Threading Technology, bit 0 is always set to 0; for Intel Xeon processors supporting Intel Hyper-Threading Technology, bit 0 performs the same function as it does for Intel Xeon processor MP.

For more recent multi-core processors, see Section 8.9.1, "Hierarchical Mapping of Shared Resources" for a complete description of the topological relationships between logical processors and bit field locations within an initial APIC ID across Intel 64 and IA-32 processor families.

Note the number of bit fields and the width of bit-fields are dependent on processor and platform hardware capabilities. Software should determine these at runtime. When initial APIC IDs are assigned to logical processors, the value of APIC ID assigned to a logical processor will respect the bit-field boundaries corresponding core, physical package, etc. Additional examples of the bit fields in the initial APIC ID of multi-threading capable systems are shown in Section 8.9.

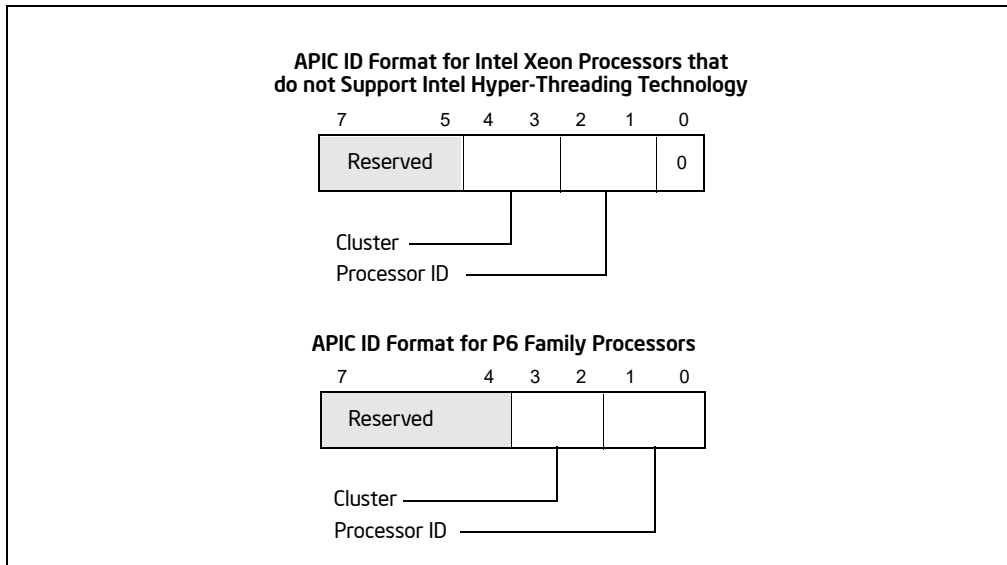


Figure 8-2. Interpretation of APIC ID in Early MP Systems

For P6 family processors, the APIC ID that is assigned to a processor during power-up and initialization is 4 bits (see Figure 8-2). Here, bits 0 and 1 form a 2-bit processor (or socket) identifier and bits 2 and 3 form a 2-bit cluster ID.

8.5 INTEL® HYPER-THREADING TECHNOLOGY AND INTEL® MULTI-CORE TECHNOLOGY

Intel Hyper-Threading Technology and Intel multi-core technology are extensions to Intel 64 and IA-32 architectures that enable a single physical processor to execute two or more separate code streams (called *threads*) concurrently. In Intel Hyper-Threading Technology, a single processor core provides two logical processors that share execution resources (see Section 8.7, “Intel® Hyper-Threading Technology Architecture”). In Intel multi-core technology, a physical processor package provides two or more processor cores. Both configurations require chipsets and a BIOS that support the technologies.

Software should not rely on processor names to determine whether a processor supports Intel Hyper-Threading Technology or Intel multi-core technology. Use the CPUID instruction to determine processor capability (see Section 8.6.2, “Initializing Multi-Core Processors”).

8.6 DETECTING HARDWARE MULTI-THREADING SUPPORT AND TOPOLOGY

Use the CPUID instruction to detect the presence of hardware multi-threading support in a physical processor. Hardware multi-threading can support several varieties of multigrade and/or Intel Hyper-Threading Technology. CPUID instruction provides several sets of parameter information to aid software enumerating topology information. The relevant topology enumeration parameters provided by CPUID include:

- **Hardware Multi-Threading feature flag (CPUID.1:EDX[28] = 1)** — Indicates when set that the physical package is capable of supporting Intel Hyper-Threading Technology and/or multiple cores.

- **Processor topology enumeration parameters for 8-bit APIC ID:**
 - **Addressable IDs for Logical processors in the same Package (CPUID.1:EBX[23:16])** — Indicates the maximum number of addressable ID for logical processors in a physical package. Within a physical package, there may be addressable IDs that are not occupied by any logical processors. This parameter does not represent the hardware capability of the physical processor.⁶
- **Addressable IDs for processor cores in the same Package⁷ (CPUID.(EAX=4, ECX=0⁸):EAX[31:26] + 1 = Y)** — Indicates the maximum number of addressable IDs attributable to processor cores (Y) in the physical package.
- **Extended Processor Topology Enumeration parameters for 32-bit APIC ID:** Intel 64 processors supporting CPUID leaf 0BH will assign unique APIC IDs to each logical processor in the system. CPUID leaf 0BH reports the 32-bit APIC ID and provide topology enumeration parameters. See CPUID instruction reference pages in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The CPUID feature flag may indicate support for hardware multi-threading when only one logical processor available in the package. In this case, the decimal value represented by bits 16 through 23 in the EBX register will have a value of 1.

Software should note that the number of logical processors enabled by system software may be less than the value of "Addressable IDs for Logical processors". Similarly, the number of cores enabled by system software may be less than the value of "Addressable IDs for processor cores".

Software can detect the availability of the CPUID extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal to 11 (0BH), then proceed to next step,
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. Note the presence of CPUID leaf 0BH in a processor does not guarantee support that the local APIC supports x2APIC. If CPUID.(EAX=0BH, ECX=0H):EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

8.6.1 Initializing Processors Supporting Hyper-Threading Technology

The initialization process for an MP system that contains processors supporting Intel Hyper-Threading Technology is the same as for conventional MP systems (see Section 8.4, "Multiple-Processor (MP) Initialization"). One logical processor in the system is selected as the BSP and other processors (or logical processors) are designated as APs. The initialization process is identical to that described in Section 8.4.3, "MP Initialization Protocol Algorithm for MP Systems," and Section 8.4.4, "MP Initialization Example."

During initialization, each logical processor is assigned an APIC ID that is stored in the local APIC ID register for each logical processor. If two or more processors supporting Intel Hyper-Threading Technology are present, each logical processor on the system bus is assigned a unique ID (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System"). Once logical processors have APIC IDs, software communicates with them by sending APIC IPI messages.

6. Operating system and BIOS may implement features that reduce the number of logical processors available in a platform to applications at runtime to less than the number of physical packages times the number of hardware-capable logical processors per package.

7. Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software should handle the situation as if there is only one core per package.

8. Maximum number of cores in the physical package must be queried by executing CPUID with EAX=4 and a valid ECX input value. Valid ECX input values start from 0.

8.6.2 Initializing Multi-Core Processors

The initialization process for an MP system that contains multi-core Intel 64 or IA-32 processors is the same as for conventional MP systems (see Section 8.4, “Multiple-Processor (MP) Initialization”). A logical processor in one core is selected as the BSP; other logical processors are designated as APs.

During initialization, each logical processor is assigned an APIC ID. Once logical processors have APIC IDs, software may communicate with them by sending APIC IPI messages.

8.6.3 Executing Multiple Threads on an Intel® 64 or IA-32 Processor Supporting Hardware Multi-Threading

Upon completing the operating system boot-up procedure, the bootstrap processor (BSP) executes operating system code. Other logical processors are placed in the halt state. To execute a code stream (thread) on a halted logical processor, the operating system issues an interprocessor interrupt (IPI) addressed to the halted logical processor. In response to the IPI, the processor wakes up and begins executing the code identified by the vector received as part of the IPI.

To manage execution of multiple threads on logical processors, an operating system can use conventional symmetric multiprocessing (SMP) techniques. For example, the operating-system can use a time-slice or load balancing mechanism to periodically interrupt each of the active logical processors. Upon interrupting a logical processor, the operating system checks its run queue for a thread waiting to be executed and dispatches the thread to the interrupted logical processor.

8.6.4 Handling Interrupts on an IA-32 Processor Supporting Hardware Multi-Threading

Interrupts are handled on processors supporting Intel Hyper-Threading Technology as they are on conventional MP systems. External interrupts are received by the I/O APIC, which distributes them as interrupt messages to specific logical processors (see Figure 8-3).

Logical processors can also send IPIs to other logical processors by writing to the ICR register of its local APIC (see Section 10.6, “Issuing Interprocessor Interrupts”). This also applies to dual-core processors.

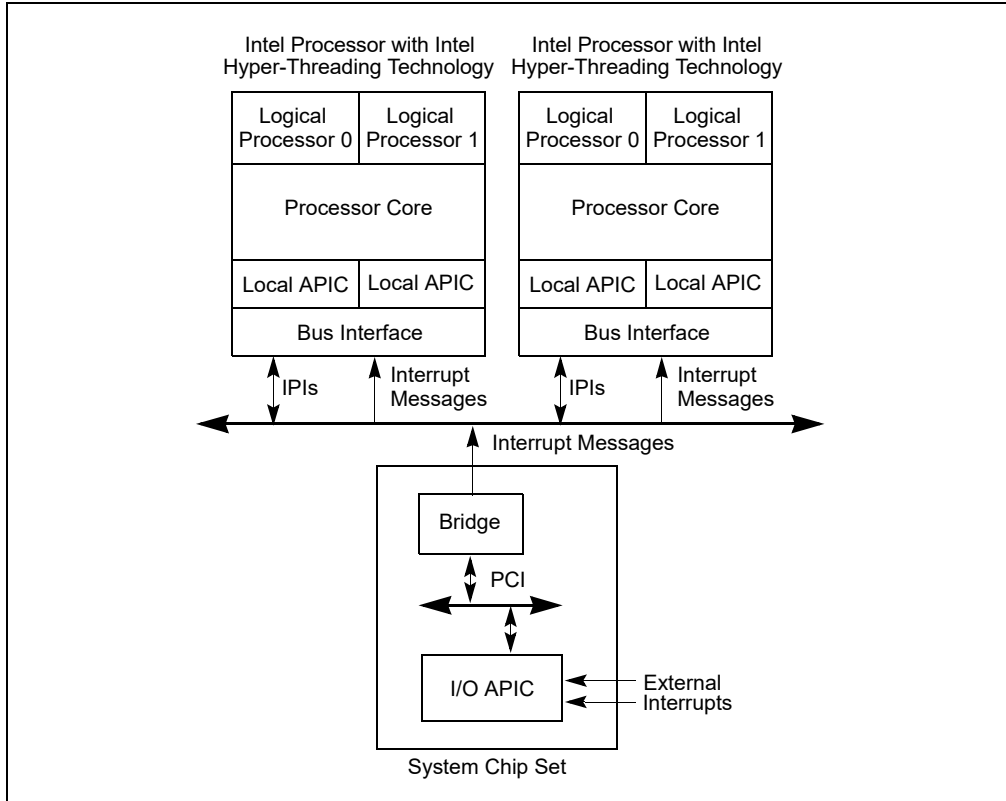


Figure 8-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology

8.7 INTEL® HYPER-THREADING TECHNOLOGY ARCHITECTURE

Figure 8-4 shows a generalized view of an Intel processor supporting Intel Hyper-Threading Technology, using the original Intel Xeon processor MP as an example. This implementation of the Intel Hyper-Threading Technology consists of two logical processors (each represented by a separate architectural state) which share the processor's execution engine and the bus interface. Each logical processor also has its own advanced programmable interrupt controller (APIC).

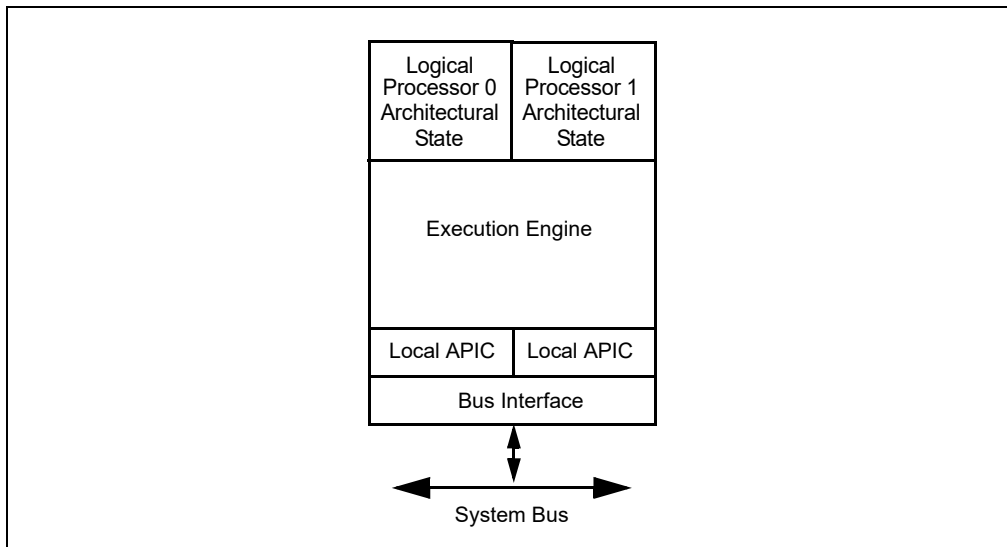


Figure 8-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology

8.7.1 State of the Logical Processors

The following features are part of the architectural state of logical processors within Intel 64 or IA-32 processors supporting Intel Hyper-Threading Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor.
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32_MCG_STATUS) and machine check capability (IA32_MCG_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32_EFER on Intel 64 processors.

The following features are shared by logical processors:

- Memory type range registers (MTRRs)

Whether the following features are shared or duplicated is implementation-specific:

- IA32_MISC_ENABLE MSR (MSR address 1A0H)

- Machine check architecture (MCA) MSR (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs)
- Performance monitoring control and counter MSRs

8.7.2 APIC Functionality

When a processor supporting Intel Hyper-Threading Technology support is initialized, each logical processor is assigned a local APIC ID (see Table 10-1). The local APIC ID serves as an ID for the logical processor and is stored in the logical processor's APIC ID register. If two or more processors supporting Intel Hyper-Threading Technology are present in a dual processor (DP) or MP system, each logical processor on the system bus is assigned a unique local APIC ID (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System").

Software communicates with local processors using the APIC's interprocessor interrupt (IPI) messaging facility. Setup and programming for APICs is identical in processors that support and do not support Intel Hyper-Threading Technology. See Chapter 10, "Advanced Programmable Interrupt Controller (APIC)," for a detailed discussion.

8.7.3 Memory Type Range Registers (MTRR)

MTRRs in a processor supporting Intel Hyper-Threading Technology are shared by logical processors. When one logical processor updates the setting of the MTRRs, settings are automatically shared with the other logical processors in the same physical package.

The architectures require that all MP systems based on Intel 64 and IA-32 processors (this includes logical processors) must use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running. See Section 11.11, "Memory Type Range Registers (MTRRs)," for information on setting up MTRRs.

8.7.4 Page Attribute Table (PAT)

Each logical processor has its own PAT MSR (IA32_PAT). However, as described in Section 11.12, "Page Attribute Table (PAT)," the PAT MSR settings must be the same for all processors in a system, including the logical processors.

8.7.5 Machine Check Architecture

In the Intel HT Technology context as implemented by processors based on Intel NetBurst[®] microarchitecture, all of the machine check architecture (MCA) MSR (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs) are duplicated for each logical processor. This permits logical processors to initialize, configure, query, and handle machine-check exceptions simultaneously within the same physical processor. The design is compatible with machine check exception handlers that follow the guidelines given in Chapter 15, "Machine-Check Architecture."

The IA32_MCG_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.

On Intel Atom family processors that support Intel Hyper-Threading Technology, the MCA facilities are shared between all logical processors on the same processor core.

8.7.6 Debug Registers and Extensions

Each logical processor has its own set of debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and its own debug control MSR. These can be set to control and record debug information for each logical processor independently. Each logical processor also has its own last branch records (LBR) stack.

8.7.7 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between the logical processors within a processor core for processors based on Intel NetBurst microarchitecture. As a result, software must manage the use of these resources. The performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 18.6.4, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst[®] Microarchitecture," for a discussion of performance monitoring in the Intel Xeon processor MP.

In Intel Atom processor family that support Intel Hyper-Threading Technology, the performance counters (general-purpose and fixed-function counters) and their companion control MSRs are duplicated for each logical processor.

8.7.8 IA32_MISC_ENABLE MSR

The IA32_MISC_ENABLE MSR (MSR address 1A0H) is generally shared between the logical processors in a processor core supporting Intel Hyper-Threading Technology. However, some bit fields within IA32_MISC_ENABLE MSR may be duplicated per logical processor. The partition of shared or duplicated bit fields within IA32_MISC_ENABLE is implementation dependent. Software should program duplicated fields carefully on all logical processors in the system to ensure consistent behavior.

8.7.9 Memory Ordering

The logical processors in an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology obey the same rules for memory ordering as Intel 64 or IA-32 processors without Intel HT Technology (see Section 8.2, "Memory Ordering"). Each logical processor uses a processor-ordered memory model that can be further defined as "write-ordered with store buffer forwarding." All mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations apply to each logical processor.

8.7.10 Serializing Instructions

As a general rule, when a logical processor in a processor supporting Intel Hyper-Threading Technology executes a serializing instruction, only that logical processor is affected by the operation. An exception to this rule is the execution of the WBINVD, INVD, and WRMSR instructions; and the MOV CR instruction when the state of the CD flag in control register CR0 is modified. Here, both logical processors are serialized.

8.7.11 Microcode Update Resources

In an Intel processor supporting Intel Hyper-Threading Technology, the microcode update facilities are shared between the logical processors; either logical processor can initiate an update. Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information. If logical processors initiate an update simultaneously, the processor core provides the necessary synchronization needed to ensure that only one update is performed at a time.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

8.7.12 Self Modifying Code

Intel processors supporting Intel Hyper-Threading Technology support self-modifying code, where data writes modify instructions cached or currently in flight. They also support cross-modifying code, where on an MP system writes generated by one processor modify instructions cached or currently in flight on another. See Section 8.1.3, “Handling Self- and Cross-Modifying Code,” for a description of the requirements for self- and cross-modifying code in an IA-32 processor.

8.7.13 Implementation-Specific Intel HT Technology Facilities

The following non-architectural facilities are implementation-specific in IA-32 processors supporting Intel Hyper-Threading Technology:

- Caches
- Translation lookaside buffers (TLBs)
- Thermal monitoring facilities

The Intel Xeon processor MP implementation is described in the following sections.

8.7.13.1 Processor Caches

For processors supporting Intel Hyper-Threading Technology, the caches are shared. Any cache manipulation instruction that is executed on one logical processor has a global effect on the cache hierarchy of the physical processor. Note the following:

- **WBINVD instruction** — The entire cache hierarchy is invalidated after modified data is written back to memory. All logical processors are stopped from executing until after the write-back and invalidate operation is completed. A special bus cycle is sent to all caching agents. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- **INVD instruction** — The entire cache hierarchy is invalidated without writing back modified data to memory. All logical processors are stopped from executing until after the invalidate operation is completed. A special bus cycle is sent to all caching agents.
- **CLFLUSH and CLFLUSHOPT instructions** — The specified cache line is invalidated from the cache hierarchy after any modified data is written back to memory and a bus cycle is sent to all caching agents, regardless of which logical processor caused the cache line to be filled.
- **CD flag in control register CR0** — Each logical processor has its own CR0 control register, and thus its own CD flag in CR0. The CD flags for the two logical processors are ORed together, such that when any logical processor sets its CD flag, the entire cache is nominally disabled.

8.7.13.2 Processor Translation Lookaside Buffers (TLBs)

In processors supporting Intel Hyper-Threading Technology, data cache TLBs are shared. The instruction cache TLB may be duplicated or shared in each logical processor, depending on implementation specifics of different processor families.

Entries in the TLBs are tagged with an ID that indicates the logical processor that initiated the translation. This tag applies even for translations that are marked global using the page-global feature for memory paging. See Section 4.10, “Caching Translation Information,” for information about global translations.

When a logical processor performs a TLB invalidation operation, only the TLB entries that are tagged for that logical processor are guaranteed to be flushed. This protocol applies to all TLB invalidation operations, including writes to control registers CR3 and CR4 and uses of the INVLPG instruction.

8.7.13.3 Thermal Monitor

In a processor that supports Intel Hyper-Threading Technology, logical processors share the catastrophic shutdown detector and the automatic thermal monitoring mechanism (see Section 14.8, “Thermal Monitoring and Protection”). Sharing results in the following behavior:

- If the processor’s core temperature rises above the preset catastrophic shutdown temperature, the processor core halts execution, which causes both logical processors to stop execution.
- When the processor’s core temperature rises above the preset automatic thermal monitor trip temperature, the frequency of the processor core is automatically modulated, which effects the execution speed of both logical processors.

For software controlled clock modulation, each logical processor has its own IA32_CLOCK_MODULATION MSR, allowing clock modulation to be enabled or disabled on a logical processor basis. Typically, if software controlled clock modulation is going to be used, the feature must be enabled for all the logical processors within a physical processor and the modulation duty cycle must be set to the same value for each logical processor. If the duty cycle values differ between the logical processors, the processor clock will be modulated at the highest duty cycle selected.

8.7.13.4 External Signal Compatibility

This section describes the constraints on external signals received through the pins of a processor supporting Intel Hyper-Threading Technology and how these signals are shared between its logical processors.

- **STPCLK#** — A single STPCLK# pin is provided on the physical package of the Intel Xeon processor MP. External control logic uses this pin for power management within the system. When the STPCLK# signal is asserted, the processor core transitions to the stop-grant state, where instruction execution is halted but the processor core continues to respond to snoop transactions. Regardless of whether the logical processors are active or halted when the STPCLK# signal is asserted, execution is stopped on both logical processors and neither will respond to interrupts.

In MP systems, the STPCLK# pins on all physical processors are generally tied together. As a result this signal affects all the logical processors within the system simultaneously.

- **LINT0 and LINT1 pins** — A processor supporting Intel Hyper-Threading Technology has only one set of LINT0 and LINT1 pins, which are shared between the logical processors. When one of these pins is asserted, both logical processors respond unless the pin has been masked in the APIC local vector tables for one or both of the logical processors.

Typically in MP systems, the LINT0 and LINT1 pins are not used to deliver interrupts to the logical processors. Instead all interrupts are delivered to the local processors through the I/O APIC.

- **A20M# pin** — On an IA-32 processor, the A20M# pin is typically provided for compatibility with the Intel 286 processor. Asserting this pin causes bit 20 of the physical address to be masked (forced to zero) for all external bus memory accesses. Processors supporting Intel Hyper-Threading Technology provide one A20M# pin, which affects the operation of both logical processors within the physical processor.

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.

8.8 MULTI-CORE ARCHITECTURE

This section describes the architecture of Intel 64 and IA-32 processors supporting dual-core and quad-core technology. The discussion is applicable to the Intel Pentium processor Extreme Edition, Pentium D, Intel Core Duo, Intel Core 2 Duo, Dual-core Intel Xeon processor, Intel Core 2 Quad processors, and quad-core Intel Xeon processors. Features vary across different microarchitectures and are detectable using CPUID.

In general, each processor core has dedicated microarchitectural resources identical to a single-processor implementation of the underlying microarchitecture without hardware multi-threading capability. Each logical processor in a dual-core processor (whether supporting Intel Hyper-Threading Technology or not) has its own APIC functionality, PAT, machine check architecture, debug registers and extensions. Each logical processor handles serialization instructions or self-modifying code on its own. Memory order is handled the same way as in Intel Hyper-Threading Technology.

The topology of the cache hierarchy (with respect to whether a given cache level is shared by one or more processor cores or by all logical processors in the physical package) depends on the processor implementation. Software must use the deterministic cache parameter leaf of CPUID instruction to discover the cache-sharing topology between the logical processors in a multi-threading environment.

8.8.1 Logical Processor Support

The topological composition of processor cores and logical processors in a multi-core processor can be discovered using CPUID. Within each processor core, one or more logical processors may be available.

System software must follow the requirement MP initialization sequences (see Section 8.4, “Multiple-Processor (MP) Initialization”) to recognize and enable logical processors. At runtime, software can enumerate those logical processors enabled by system software to identify the topological relationships between these logical processors. (See Section 8.9.5, “Identifying Topological Relationships in a MP System”).

8.8.2 Memory Type Range Registers (MTRR)

MTRR is shared between two logical processors sharing a processor core if the physical processor supports Intel Hyper-Threading Technology. MTRR is not shared between logical processors located in different cores or different physical packages.

The Intel 64 and IA-32 architectures require that all logical processors in an MP system use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running.

See Section 11.11, “Memory Type Range Registers (MTRRs).”

8.8.3 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between two logical processors sharing a processor core if the processor core supports Intel Hyper-Threading Technology and is based on Intel NetBurst microarchitecture. They are not shared between logical processors in different cores or different physical packages. As a result, software must manage the use of these resources, based on the topology of performance monitoring resources. Performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 18.6.4, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture.”

8.8.4 IA32_MISC_ENABLE MSR

Some bit fields in IA32_MISC_ENABLE MSR (MSR address 1A0H) may be shared between two logical processors sharing a processor core, or may be shared between different cores in a physical processor. See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

8.8.5 Microcode Update Resources

Microcode update facilities are shared between two logical processors sharing a processor core if the physical package supports Intel Hyper-Threading Technology. They are not shared between logical processors in different

cores or different physical packages. Either logical processor that has access to the microcode update facility can initiate an update.

Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information.

All microcode update steps during processor initialization should use the same update data on all cores in all physical packages of the same stepping. Any subsequent microcode update must apply consistent update data to all cores in all physical packages of the same stepping. If the processor detects an attempt to load an older microcode update when a newer microcode update had previously been loaded, it may reject the older update to stay with the newer update.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

8.9 PROGRAMMING CONSIDERATIONS FOR HARDWARE MULTI-THREADING CAPABLE PROCESSORS

In a multi-threading environment, there may be certain hardware resources that are physically shared at some level of the hardware topology. In the multi-processor systems, typically bus and memory sub-systems are physically shared between multiple sockets. Within a hardware multi-threading capable processors, certain resources are provided for each processor core, while other resources may be provided for each logical processors (see Section 8.7, “Intel® Hyper-Threading Technology Architecture,” and Section 8.8, “Multi-Core Architecture”).

From a software programming perspective, control transfer of processor operation is managed at the granularity of logical processor (operating systems dispatch a runnable task by allocating an available logical processor on the platform). To manage the topology of shared resources in a multi-threading environment, it may be useful for software to understand and manage resources that are shared by more than one logical processors.

8.9.1 Hierarchical Mapping of Shared Resources

The APIC_ID value associated with each logical processor in a multi-processor system is unique (see Section 8.6, “Detecting Hardware Multi-Threading Support and Topology”). This 8-bit or 32-bit value can be decomposed into sub-fields, where each sub-field corresponds a hierarchical level of the topological mapping of hardware resources.

The decomposition of an APIC_ID may consist of several sub fields representing the topology within a physical processor package, the higher-order bits of an APIC ID may also be used by cluster vendors to represent the topology of cluster nodes of each coherent multiprocessor systems:

- **Cluster** — Some multi-threading environments consists of multiple clusters of multi-processor systems. The CLUSTER_ID sub-field is usually supported by vendor firmware to distinguish different clusters. For non-clustered systems, CLUSTER_ID is usually 0 and system topology is reduced.
- **Package** — A physical processor package mates with a socket. A package may contain one or more software visible die. The PACKAGE_ID sub-field distinguishes different physical packages within a cluster.
- **Die** — A software-visible chip inside a package. The DIE_ID sub-field distinguishes different die within a package. If there are no software visible die, the width of this bit field is 0.
- **Tile** — A set of cores, possibly within modules, that share certain resources. The TILE_ID sub-field distinguishes different tiles. If there are no software visible tiles, the width of this bit field is 0.
- **Module** — A set of cores that share certain resources. The MODULE_ID sub-field distinguishes different modules. If there are no software visible modules, the width of this bit field is 0.

- **Core** — Processor cores may be contained within modules, within tiles, on software-visible die, or appear directly at the package level. The CORE_ID sub-field distinguishes processor cores. For a single-core processor, the width of this bit field is 0.
- **SMT** — A processor core provides one or more logical processors sharing execution resources. The SMT_ID sub-field distinguishes logical processors in a core. The width of this bit field is non-zero if a processor core provides more than one logical processors.

SMT and CORE sub-fields are bit-wise contiguous in the APIC_ID field (see Figure 8-5).

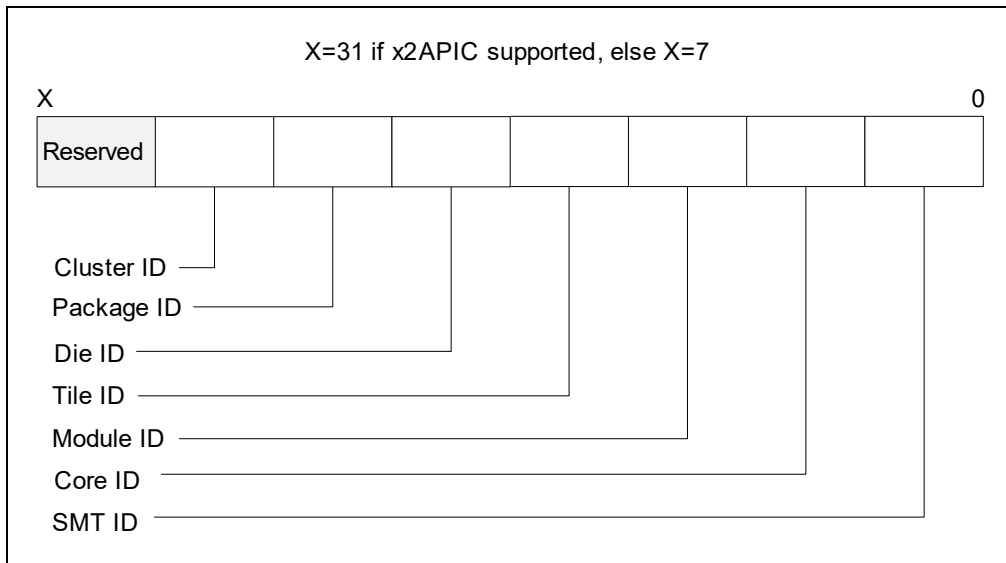


Figure 8-5. Generalized Seven Level Interpretation of the APIC ID

If the processor supports CPUID leaf 0BH and leaf 1FH, the 32-bit APIC ID can represent cluster plus several levels of topology within the physical processor package. The exact number of hierarchical levels within a physical processor package must be enumerated through CPUID leaf 0BH and leaf 1FH. Common processor families may employ topology similar to that represented by 8-bit Initial APIC ID. In general, CPUID leaf 0BH and leaf 1FH can support a topology enumeration algorithm that decompose a 32-bit APIC ID into more than four sub-fields (see Figure 8-6).

NOTE

CPUID leaf 0BH and leaf 1FH can have differences in the number of level types reported (CPUID leaf 1FH defines additional level types). If the processor supports CPUID leaf 1FH, usage of this leaf is preferred over leaf 0BH. CPUID leaf 0BH is available for legacy compatibility going forward.

The width of each sub-field depends on hardware and software configurations. Field widths can be determined at runtime using the algorithm discussed below (Example 8-16 through Example 8-21).

Figure 7-6 depicts the relationships of three of the hierarchical sub-fields in a hypothetical MP system. The value of valid APIC_IDs need not be contiguous across package boundary or core boundaries.

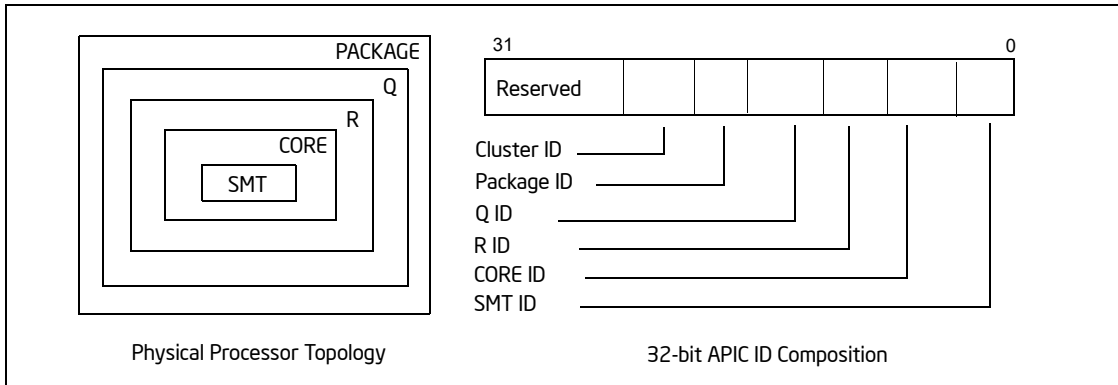


Figure 8-6. Conceptual Six-Level Topology and 32-bit APIC ID Composition

8.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf

CPUID leaf 0BH and leaf 1FH provide enumeration parameters for software to identify each hierarchy of the processor topology in a deterministic manner. Each hierarchical level of the topology starting from the SMT level is represented numerically by a sub-leaf index within the CPUID 0BH leaf and 1FH leaf. Each level of the topology is mapped to a sub-field in the APIC ID, following the general relationship depicted in Figure 8-6. This mechanism allows software to query the exact number of levels within a physical processor package and the bit-width of each sub-field of x2APIC ID directly. For example,

- Starting from sub-leaf index 0 and incrementing ECX until CPUID.(EAX=0BH or 1FH, ECX=N):ECX[15:8] returns an invalid "level type" encoding. The number of levels within the physical processor package is "N" (excluding PACKAGE). Using Figure 8-6 as an example, CPUID.(EAX=0BH or 1FH, ECX=4):ECX[15:8] will report 00H, indicating sub leaf 04H is invalid. This is also depicted by a pseudo code example:

Example 8-16. Number of Levels Below the Physical Processor Package

```

Byte type = 1;
s = 0;
While ( type ) {
    EAX = 0BH or 1FH; // query each sub leaf of CPUID leaf 0BH or 1FH; CPUID leaf 1FH is preferred over leaf 0BH if available
    ECX = s;
    CPUID;
    type = ECX[15:8]; // examine level type encoding
    s ++;
}

```

N = ECX[7:0];

- Sub-leaf index 0 (ECX= 0 as input) provides enumeration parameters to extract the SMT sub-field of x2APIC ID. If EAX = 0BH or 1FH, and ECX =0 is specified as input when executing CPUID, CPUID.(EAX=0BH or 1FH, ECX=0):EAX[4:0] reports a value (a right-shift count) that allow software to extract part of x2APIC ID to distinguish the next higher topological entities above the SMT level. This value also corresponds to the bit-width of the sub-field of x2APIC ID corresponding the hierarchical level with sub-leaf index 0.
- For each subsequent higher sub-leaf index m, CPUID.(EAX=0BH or 1FH, ECX=m):EAX[4:0] reports the right-shift count that will allow software to extract part of x2APIC ID to distinguish higher-level topological entities. This means the right-shift value at of sub-leaf m, corresponds to the least significant (m+1) subfields of the 32-bit x2APIC ID.

Example 8-17. BitWidth Determination of x2APIC ID Subfields

```

For m = 0, m < N, m ++;
{ cumulative_width[m] = CPUID.(EAX=0BH or 1FH, ECX= m): EAX[4:0]; }
BitWidth[0] = cumulative_width[0];
For m = 1, m < N, m ++;
  BitWidth[m] = cumulative_width[m] - cumulative_width[m-1];

```

NOTE

CPUID leaf 1FH is a preferred superset to leaf 0BH. Leaf 1FH defines additional level types, and it must be parsed by an algorithm that can handle the addition of future level types.

Previously, only the following encoding of hierarchical level types were defined: 0 (invalid), 1 (SMT), and 2 (core). With the additional hierarchical level types available (see Section 8.9.1, “Hierarchical Mapping of Shared Resources” and Figure 8-5, “Generalized Seven Level Interpretation of the APIC ID”) software must not assume any “level type” encoding value to be related to any sub-leaf index, except sub-leaf 0.

Example 8-18. Support Routines for Identifying Package, Die, Core and Logical Processors from 32-bit x2APIC ID**a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.**

```

//
// This example shows how to enumerate CPU topology level types (level types may or may not be known/supported by the software)
//
// Below is the list of sample level types used in the example.
// Refer to the CPUID Leaf 1FH definition for the actual level type numbers: “V2 Extended Topology Enumeration Leaf”.
//
// SMT
// CORE
// MODULE
// TILE
// DIE
// PACKAGE
//
// The example shows how to identify and derive the extraction bitmask for the levels with identify type SMT/CORE/DIE/PACKAGE
//

```

```

int DeriveSMT_Mask_Offsets (void)
{
  IF (!HWMTSupported()) return -1;
  execute cpuid with EAX = 0BH or 1FH, ECX = 0;
  IF (returned level type encoding in ECX[15:8] does not match SMT) return -1;
  Mask_SMT_shift = EAX[4:0]; // # bits shift right of APIC ID to distinguish different cores
  SMT_MASK = ~( -1 ) << Mask_SMT_shift; // shift left to derive extraction bitmask for SMT_ID
  return 0;
}

```

- b. Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.

```

int DeriveCore_Mask_Offsets (void)
{
    IF (!HWMTSupported()) return -1;
    execute cpuid with EAX = 0BH or 1FH, ECX = 0;
    WHILE( ECX[15:8] ) { //level type encoding is valid
        Mask_last_known_shift = EAX[4:0]
        IF (returned level type encoding in ECX[15:8] matches CORE) {
            Mask_Core_shift = EAX[4:0];
        }
        ELSE IF (returned level type encoding in ECX[15:8] matches DIE {
            Mask_Die_shift = EAX[4:0];
        }
        //
        // Keep enumerating. Check if the next level is the desired level and if not, keep enumerating until you reach a known level
        // or the invalid level ("0" level type). If there are more levels between DIE and PACKAGE, the unknown levels will be ignored
        // and treated as an extension of the last known level (i.e., DIE in this case).
        //

        ECX++;
        execute cpuid with EAX = 0BH or 1FH;
    }

    COREPlusSMT_MASK = ~( (-1) << Mask_Core_shift);
    DIEPlusCORE_MASK = ~( (-1) << Mask_Die_shift);

    //
    // Treat levels between DIE and physical package as an extension of DIE for software choosing not to implement or recognize
    // these unknown levels.
    //

    CORE_MASK = COREPlusSMT_MASK ^ SMT_MASK;
    DIE_MASK = DIEPlusCORE_MASK ^ COREPlusSMT_MASK;
    PACKAGE_MASK = (-1) << Mask_last_known_shift;

    return -1;
}

```

8.9.3 Hierarchical ID of Logical Processors in an MP System

For Intel 64 and IA-32 processors, system hardware establishes an 8-bit initial APIC ID (or 32-bit APIC ID if the processor supports CPUID leaf 0BH) that is unique for each logical processor following power-up or RESET (see Section 8.6.1). Each logical processor on the system is allocated an initial APIC ID. BIOS may implement features that tell the OS to support less than the total number of logical processors on the system bus. Those logical processors that are not available to applications at runtime are halted during the OS boot process. As a result, the number valid local APIC_IDs that can be queried by affinityizing-current-thread-context (See Example 8-23) is limited to the number of logical processors enabled at runtime by the OS boot process.

Table 8-2 shows an example of the 8-bit APIC IDs that are initially reported for logical processors in a system with four Intel Xeon MP processors that support Intel Hyper-Threading Technology (a total of 8 logical processors, each physical package has two processor cores and supports Intel Hyper-Threading Technology). Of the two logical

processors within a Intel Xeon processor MP, logical processor 0 is designated the primary logical processor and logical processor 1 as the secondary logical processor.

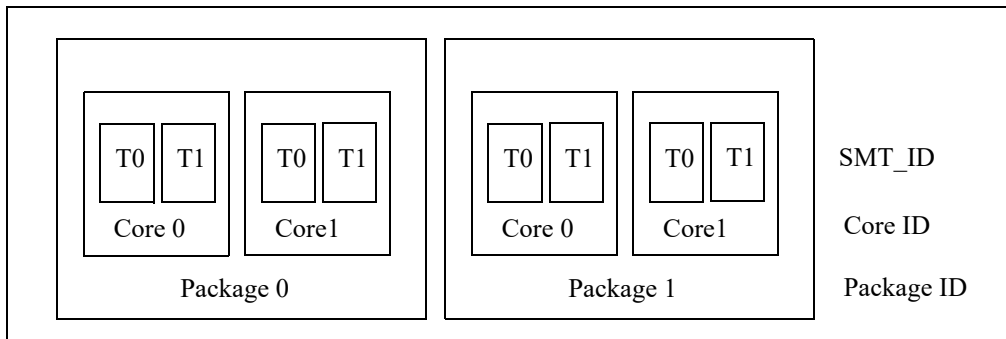


Figure 8-7. Topological Relationships between Hierarchical IDs in a Hypothetical MP Platform

Table 8-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology¹

Initial APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	1H	0H	0H
3H	1H	0H	1H
4H	2H	0H	0H
5H	2H	0H	1H
6H	3H	0H	0H
7H	3H	0H	1H

NOTE:

1. Because information on the number of processor cores in a physical package was not available in early single-core processors supporting Intel Hyper-Threading Technology, the core ID can be treated as 0.

Table 8-3 shows the initial APIC IDs for a hypothetical situation with a dual processor system. Each physical package providing two processor cores, and each processor core also supporting Intel Hyper-Threading Technology.

Table 8-3. Initial APIC IDs for the Logical Processors in a System that has Two Physical Processors Supporting Dual-Core and Intel Hyper-Threading Technology

Initial APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	1H	0H	0H
5H	1H	0H	1H
6H	1H	1H	0H
7H	1H	1H	1H

8.9.3.1 Hierarchical ID of Logical Processors with x2APIC ID

Table 8-4 shows an example of possible x2APIC ID assignments for a dual processor system that support x2APIC. Each physical package providing four processor cores, and each processor core also supporting Intel Hyper-Threading Technology. Note that the x2APIC ID need not be contiguous in the system.

Table 8-4. Example of Possible x2APIC ID Assignment in a System that has Two Physical Processors Supporting x2APIC and Intel Hyper-Threading Technology

x2APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	0H	2H	0H
5H	0H	2H	1H
6H	0H	3H	0H
7H	0H	3H	1H
10H	1H	0H	0H
11H	1H	0H	1H
12H	1H	1H	0H
13H	1H	1H	1H
14H	1H	2H	0H
15H	1H	2H	1H
16H	1H	3H	0H
17H	1H	3H	1H

8.9.4 Algorithm for Three-Level Mappings of APIC_ID

Software can gather the initial APIC_IDs for each logical processor supported by the operating system at runtime⁹ and extract identifiers corresponding to the three levels of sharing topology (package, core, and SMT). The three-level algorithms below focus on a non-clustered MP system for simplicity. They do not assume APIC IDs are contiguous or that all logical processors on the platform are enabled.

Intel supports multi-threading systems where all physical processors report identical values in CPUID leaf 0BH, CPUID.1:EBX[23:16]), CPUID.4¹⁰:EAX[31:26], and CPUID.4¹¹:EAX[25:14]. The algorithms below assume the target system has symmetry across physical package boundaries with respect to the number of logical processors per package, number of cores per package, and cache topology within a package.

Software can choose to assume three level hierarchy if it was developed to understand only three levels. However, software implementation needs to ensure it does not break if it runs on systems that have more levels in the hierarchy even if it does not recognize them.

-
- 9. As noted in Section 8.6 and Section 8.9.3, the number of logical processors supported by the OS at runtime may be less than the total number logical processors available in the platform hardware.
 - 10. Maximum number of addressable ID for processor cores in a physical processor is obtained by executing CPUID with EAX=4 and a valid ECX index, The ECX index start at 0.
 - 11. Maximum number addressable ID for processor cores sharing the target cache level is obtained by executing CPUID with EAX = 4 and the ECX index corresponding to the target cache level.

The extraction algorithm (for three-level mappings from an APIC ID) uses the general procedure depicted in Example 8-19, and is supplemented by more detailed descriptions on the derivation of topology enumeration parameters for extraction bit masks:

1. Detect hardware multi-threading support in the processor.
2. Derive a set of bit masks that can extract the sub ID of each hierarchical level of the topology. The algorithm to derive extraction bit masks for SMT_ID/CORE_ID/PACKAGE_ID differs based on APIC ID is 32-bit (see step 3 below) or 8-bit (see step 4 below):
3. If the processor supports CPUID leaf 0BH, each APIC ID contains a 32-bit value, the topology enumeration parameters needed to derive three-level extraction bit masks are:
 - a. Query the right-shift value for the SMT level of the topology using CPUID leaf 0BH with ECX = 0H as input. The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-level entities above SMT (e.g. processor cores) in the same physical package. This is also the width of the bit mask to extract the SMT_ID.
 - b. Enumerate until the desired level is found (i.e. processor cores). Determine if the next level is the expected level. If the next level is not known to the software, keep enumerating until the next known or the last level. Software should use the previous level before this to represent the last previously known level (i.e. processor cores). If the software does not recognize or implement certain hierarchical levels, it should assume these unknown levels as an extension of the last known level.
 - c. Query CPUID leaf 0BH for the amount of bit shift to distinguish next higher-level entities (e.g. physical processor packages) in the system. This describes an explicit three-level-topology situation for commonly available processors. Consult Example 8-17 to adapt to situations beyond three-level topology of a physical processor. The width of the extraction bit mask can be used to derive the cumulative extraction bitmask to extract the sub IDs of logical processors (including different processor cores) in the same physical package. The extraction bit mask to distinguish merely different processor cores can be derived by xor'ing the SMT extraction bit mask from the cumulative extraction bit mask.
 - d. Query the 32-bit x2APIC ID for the logical processor where the current thread is executing.
 - e. Derive the extraction bit masks corresponding to SMT_ID, CORE_ID, and PACKAGE_ID, starting from SMT_ID.
 - f. Apply each extraction bit mask to the 32-bit x2APIC ID to extract sub-field IDs.
4. If the processor does not support CPUID leaf 0BH, each initial APIC ID contains an 8-bit value, the topology enumeration parameters needed to derive extraction bit masks are:
 - a. Query the size of address space for sub IDs that can accommodate logical processors in a physical processor package. This size parameters (CPUID.1:EBX[23:16]) can be used to derive the width of an extraction bitmask to enumerate the sub IDs of different logical processors in the same physical package.
 - b. Query the size of address space for sub IDs that can accommodate processor cores in a physical processor package. This size parameters can be used to derive the width of an extraction bitmask to enumerate the sub IDs of processor cores in the same physical package.
 - c. Query the 8-bit initial APIC ID for the logical processor where the current thread is executing.
 - d. Derive the extraction bit masks using respective address sizes corresponding to SMT_ID, CORE_ID, and PACKAGE_ID, starting from SMT_ID.
 - e. Apply each extraction bit mask to the 8-bit initial APIC ID to extract sub-field IDs.

Example 8-19. Support Routines for Detecting Hardware Multi-Threading and Identifying the Relationships Between Package, Core and Logical Processors**1. Detect support for Hardware Multi-Threading Support in a processor.**

```
// Returns a non-zero value if CPUID reports the presence of hardware multi-threading
// support in the physical package where the current logical processor is located.
// This does not guarantee BIOS or OS will enable all logical processors in the physical
// package and make them available to applications.
// Returns zero if hardware multi-threading is not present.
```

```
#define HWMT_BIT 10000000H
```

```
unsigned int HWMTSupported(void)
{
    // ensure cpuid instruction is supported
    // execute cpuid with eax = 0 to get vendor string
    // execute cpuid with eax = 1 to get feature flag and signature

    // Check to see if this a Genuine Intel Processor

    if (vendor string EQ GenuineIntel) {
        return (feature_flag_edx & HWMT_BIT); // bit 28
    }
    return 0;
}
```

Example 8-20. Support Routines for Identifying Package, Core and Logical Processors from 32-bit x2APIC ID**a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.**

```
int DeriveSMT_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
    execute cpuid with eax = 11, ECX = 0;
    If (returned level type encoding in ECX[15:8] does not match SMT) return -1;
    Mask_SMT_shift = EAX[4:0]; // # bits shift right of APIC ID to distinguish different cores
    SMT_MASK = ~((-1) << Mask_SMT_shift); // shift left to derive extraction bitmask for SMT_ID
    return 0;
}
```


- b. **Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.**

```
int DeriveCore_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
    execute cpuid with eax = 11, ECX = 0;
    while( ECX[15:8] ) {          // level type encoding is valid
        Mask_Core_shift = EAX[4:0];    // needed to distinguish different physical packages
        ECX ++;
        execute cpuid with eax = 11;
    }
    COREPlusSMT_MASK = ~( (-1) << Mask_Core_shift);
    // treat levels between core and physical package as a core for software choosing not to implement or recognize
    // these unknown levels
    CORE_MASK = COREPlusSMT_MASK ^ SMT_MASK;
    PACKAGE_MASK = (-1) << Mask_Core_shift;
    return -1;
}
```

- c. **Query the x2APIC ID of a logical processor.**

APIC_IDs for each logical processor.

```
unsigned char Getx2APIC_ID (void)
{
    unsigned reg_edx = 0;
    execute cpuid with eax = 11, ECX = 0
    store returned value of edx
    return (unsigned) (reg_edx);
}
```

Example 8-21. Support Routines for Identifying Package, Core and Logical Processors from 8-bit Initial APIC ID

- a. **Find the size of address space for logical processors in a physical processor package.**

```
#define NUM_LOGICAL_BITS 00FF0000H
// Use the mask above and CPUID.1.EBX[23:16] to obtain the max number of addressable IDs
// for logical processors in a physical package,

//Returns the size of address space of logical processors in a physical processor package;
// Software should not assume the value to be a power of 2.

unsigned char MaxLPIDsPerPackage(void)
{
    if (!HWMTSupported()) return 1;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```

b. Find the size of address space for processor cores in a physical processor package.

// Returns the max number of addressable IDs for processor cores in a physical processor package;
// Software should not assume cpuid reports this value to be a power of 2.

```
unsigned MaxCoreIDsPerPackage(void)
{
    if (!HWMTSupported()) return (unsigned char) 1;
    if cpuid supports leaf number 4
    { // we can retrieve multi-core topology info using leaf 4
        execute cpuid with eax = 4, ecx = 0
        store returned value of eax
        return (unsigned) ((reg_eax >> 26) + 1);
    }
    else // must be a single-core processor
        return 1;
}
```

c. Query the initial APIC ID of a logical processor.

#define INITIAL_APIC_ID_BITS FF000000H // CPUID.1.EBX[31:24] initial APIC ID

// Returns the 8-bit unique initial APIC ID for the processor running the code.
// Software can use OS services to affinitize the current thread to each logical processor
// available under the OS to gather the initial APIC_IDs for each logical processor.

```
unsigned GetInitAPIC_ID (void)
{
    unsigned int reg_ebx = 0;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned) ((reg_ebx & INITIAL_APIC_ID_BITS) >> 24);
}
```

d. Find the width of an extraction bitmask from the maximum count of the bit-field (address size).

```
// Returns the mask bit width of a bit field from the maximum count that bit field can represent.
// This algorithm does not assume 'address size' to have a value equal to power of 2.
// Address size for SMT_ID can be calculated from MaxLPIDsPerPackage()/MaxCoreIDsPerPackage()
// Then use the routine below to derive the corresponding width of SMT extraction bitmask
// Address size for CORE_ID is MaxCoreIDsPerPackage(),
// Derive the bitwidth for CORE extraction mask similarly
```

```
unsigned FindMaskWidth(unsigned Max_Count)
{unsigned int mask_width, cnt = Max_Count;
  __asm {
    mov eax, cnt
    mov ecx, 0
    mov mask_width, ecx
    dec eax
    bsr cx, ax
    jz next
    inc cx
    mov mask_width, ecx
  next:
    mov eax, mask_width
  }
  return mask_width;
}
```

e. Extract a sub ID from an 8-bit full ID, using address size of the sub ID and shift count.

```
// The routine below can extract SMT_ID, CORE_ID, and PACKAGE_ID respectively from the init APIC_ID
// To extract SMT_ID, MaxSubIDvalue is set to the address size of SMT_ID, Shift_Count = 0
// To extract CORE_ID, MaxSubIDvalue is the address size of CORE_ID, Shift_Count is width of SMT extraction bitmask.
// Returns the value of the sub ID, this is not a zero-based value
```

```
Unsigned char GetSubID(unsigned char Full_ID, unsigned char MaxSubIDvalue, unsigned char Shift_Count)
{
  MaskWidth = FindMaskWidth(MaxSubIDvalue);
  MaskBits = ((uchar) (FFH << Shift_Count)) ^ ((uchar) (FFH << Shift_Count + MaskWidth));
  SubID = Full_ID & MaskBits;
  Return SubID;
}
```

Software must not assume local APIC_ID values in an MP system are consecutive. Non-consecutive local APIC_IDs may be the result of hardware configurations or debug features implemented in the BIOS or OS.

An identifier for each hierarchical level can be extracted from an 8-bit APIC_ID using the support routines illustrated in Example 8-21. The appropriate bit mask and shift value to construct the appropriate bit mask for each level must be determined dynamically at runtime.

8.9.5 Identifying Topological Relationships in a MP System

To detect the number of physical packages, processor cores, or other topological relationships in a MP system, the following procedures are recommended:

- Extract the three-level identifiers from the APIC ID of each logical processor enabled by system software. The sequence is as follows (See the pseudo code shown in Example 8-22 and support routines shown in Example 8-19):

- The extraction start from the right-most bit field, corresponding to SMT_ID, the innermost hierarchy in a three-level topology (See Figure 8-7). For the right-most bit field, the shift value of the working mask is zero. The width of the bit field is determined dynamically using the maximum number of logical processor per core, which can be derived from information provided from CPUID.
- To extract the next bit-field, the shift value of the working mask is determined from the width of the bit mask of the previous step. The width of the bit field is determined dynamically using the maximum number of cores per package.
- To extract the remaining bit-field, the shift value of the working mask is determined from the maximum number of logical processor per package. So the remaining bits in the APIC ID (excluding those bits already extracted in the two previous steps) are extracted as the third identifier. This applies to a non-clustered MP system, or if there is no need to distinguish between PACKAGE_ID and CLUSTER_ID.

If there is need to distinguish between PACKAGE_ID and CLUSTER_ID, PACKAGE_ID can be extracted using an algorithm similar to the extraction of CORE_ID, assuming the number of physical packages in each node of a clustered system is symmetric.

- Assemble the three-level identifiers of SMT_ID, CORE_ID, PACKAGE_IDs into arrays for each enabled logical processor. This is shown in Example 8-23a.
- To detect the number of physical packages: use PACKAGE_ID to identify those logical processors that reside in the same physical package. This is shown in Example 8-23b. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same package.
- To detect the number of processor cores: use CORE_ID to identify those logical processors that reside in the same core. This is shown in Example 8-23. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same core.

In Example 8-22, the numerical ID value can be obtained from the value extracted with the mask by shifting it right by shift count. Algorithms below do not shift the value. The assumption is that the SubID values can be compared for equivalence without the need to shift.

Example 8-22. Pseudo Code Depicting Three-level Extraction Algorithm

```

For Each local_APIC_ID{
    // Calculate SMT_MASK, the bit mask pattern to extract SMT_ID,
    // SMT_MASK is determined using topology enumertaion parameters
    // from CPUID leaf 0BH (Example 8-20);
    // otherwise, SMT_MASK is determined using CPUID leaf 01H and leaf 04H (Example 8-21).
    // This algorithm assumes there is symmetry across core boundary, i.e. each core within a
    // package has the same number of logical processors
    // SMT_ID always starts from bit 0, corresponding to the right-most bit-field
    SMT_ID = APIC_ID & SMT_MASK;

// Extract CORE_ID:
    // CORE_MASK is determined in Example 8-20 or Example 8-21
    CORE_ID = (APIC_ID & CORE_MASK);

    // Extract PACKAGE_ID:
    // Assume single cluster.
    // Shift out the mask width for maximum logical processors per package
    // PACKAGE_MASK is determined in Example 8-20 or Example 8-21
    PACKAGE_ID = (APIC_ID & PACKAGE_MASK);
}
    
```

Example 8-23. Compute the Number of Packages, Cores, and Processor Relationships in a MP System

a) Assemble lists of PACKAGE_ID, CORE_ID, and SMT_ID of each enabled logical processors

```
//The BIOS and/or OS may limit the number of logical processors available to applications
// after system boot. The below algorithm will compute topology for the processors visible
// to the thread that is computing it.

// Extract the 3-levels of IDs on every processor
// SystemAffinity is a bitmask of all the processors started by the OS. Use OS specific APIs to
// obtain it.
// ThreadAffinityMask is used to affinitize the topology enumeration thread to each processor
using OS specific APIs.
// Allocate per processor arrays to store the Package_ID, Core_ID and SMT_ID for every started
// processor.
```

```
ThreadAffinityMask = 1;
ProcessorNum = 0;
while (ThreadAffinityMask != 0 && ThreadAffinityMask <= SystemAffinity) {
    // Check to make sure we can utilize this processor first.
    if (ThreadAffinityMask & SystemAffinity){
        Set thread to run on the processor specified in ThreadAffinityMask
        Wait if necessary and ensure thread is running on specified processor

        APIC_ID = GetAPIC_ID(); // 32 bit ID in Example 8-20 or 8-bit ID in Example 8-21
        Extract the Package_ID, Core_ID and SMT_ID as explained in three level extraction
        algorithm of Example 8-22
        PackageID[ProcessorNUM] = PACKAGE_ID;
        CoreID[ProcessorNum] = CORE_ID;
        SmtID[ProcessorNum] = SMT_ID;
        ProcessorNum++;
    }
    ThreadAffinityMask <<= 1;
}
NumStartedLPs = ProcessorNum;
```

b) Using the list of PACKAGE_ID to count the number of physical packages in a MP system and construct, for each package, a multi-bit mask corresponding to those logical processors residing in the same package.

```
// Compute the number of packages by counting the number of processors
// with unique PACKAGE_IDs in the PackageID array.
// Compute the mask of processors in each package.
```

PackageIDBucket is an array of unique PACKAGE_ID values. Allocate an array of NumStartedLPs count of entries in this array.
 PackageProcessorMask is a corresponding array of the bit mask of processors belonging to the same package, these are processors with the same PACKAGE_ID
 The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.
 // Bucket Package IDs and compute processor mask for every package.

```
PackageNum = 1;
PackageIDBucket[0] = PackageID[0];
ProcessorMask = 1;
PackageProcessorMask[0] = ProcessorMask;
```

MULTIPLE-PROCESSOR MANAGEMENT

```
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask <<= 1;
    For (i=0; i < PackageNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different
        // packages, the code below assume package symmetry
        If (PackageID[ProcessorNum] = PackageIDBucket[i]) {
            PackageProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    if (i = PackageNum) {
        //PACKAGE_ID did not match any bucket, start new bucket
        PackageIDBucket[i] = PackageID[ProcessorNum];
        PackageProcessorMask[i] = ProcessorMask;
        PackageNum++;
    }
}
// PackageNum has the number of Packages started in OS
// PackageProcessorMask[] array has the processor set of each package
```

c) Using the list of CORE_ID to count the number of cores in a MP system and construct, for each core, a multi-bit mask corresponding to those logical processors residing in the same core.

Processors in the same core can be determined by bucketing the processors with the same PACKAGE_ID and CORE_ID. Note that code below can BIT OR the values of PACKAGE and CORE ID because they have not been shifted right.

The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.

```
//Bucketing PACKAGE and CORE IDs and computing processor mask for every core
CoreNum = 1;
CoreIDBucket[0] = PackageID[0] | CoreID[0];
ProcessorMask = 1;
CoreProcessorMask[0] = ProcessorMask;
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask <<= 1;
    For (i=0; i < CoreNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different
        // packages, the code below assume package symmetry
        If ((PackageID[ProcessorNum] | CoreID[ProcessorNum]) = CoreIDBucket[i]) {
            CoreProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    if (i = CoreNum) {
        //Did not match any bucket, start new bucket
        CoreIDBucket[i] = PackageID[ProcessorNum] | CoreID[ProcessorNum];
        CoreProcessorMask[i] = ProcessorMask;
        CoreNum++;
    }
}
// CoreNum has the number of cores started in the OS
// CoreProcessorMask[] array has the processor set of each core
```

Other processor relationships such as processor mask of sibling cores can be computed from set operations of the PackageProcessorMask[] and CoreProcessorMask[].

The algorithm shown above can be adapted to work with earlier generations of single-core IA-32 processors that support Intel Hyper-Threading Technology and in situations that the deterministic cache parameter leaf is not supported (provided CPUID supports initial APIC ID). A reference code example is available (see *Intel® 64 Architecture Processor Topology Enumeration*).

8.10 MANAGEMENT OF IDLE AND BLOCKED CONDITIONS

When a logical processor in an MP system (including multi-core processor or processors supporting Intel Hyper-Threading Technology) is idle (no work to do) or blocked (on a lock or semaphore), additional management of the core execution engine resource can be accomplished by using the HLT (halt), PAUSE, or the MONITOR/MWAIT instructions.

8.10.1 HLT Instruction

The HLT instruction stops the execution of the logical processor on which it is executed and places it in a halted state until further notice (see the description of the HLT instruction in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). When a logical processor is halted, active logical processors continue to have full access to the shared resources within the physical package. Here shared resources that were being used by the halted logical processor become available to active logical processors, allowing them to execute at greater efficiency. When the halted logical processor resumes execution, shared resources are again shared among all active logical processors. (See Section 8.10.6.3, "Halt Idle Logical Processors," for more information about using the HLT instruction with processors supporting Intel Hyper-Threading Technology.)

8.10.2 PAUSE Instruction

The PAUSE instruction can improve the performance of processors supporting Intel Hyper-Threading Technology when executing "spin-wait loops" and other routines where one thread is accessing a shared lock or semaphore in a tight polling loop. When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. In addition, the PAUSE instruction depipelines the spin-wait loop to prevent it from consuming execution resources excessively and consume power needlessly. (See Section 8.10.6.1, "Use the PAUSE Instruction in Spin-Wait Loops," for more information about using the PAUSE instruction with IA-32 processors supporting Intel Hyper-Threading Technology.)

8.10.3 Detecting Support MONITOR/MWAIT Instruction

Streaming SIMD Extensions 3 introduced two instructions (MONITOR and MWAIT) to help multithreaded software improve thread synchronization. In the initial implementation, MONITOR and MWAIT are available to software at ring 0. The instructions are conditionally available at levels greater than 0. Use the following steps to detect the availability of MONITOR and MWAIT:

- Use CPUID to query the MONITOR bit (CPUID.1.ECX[3] = 1).
- If CPUID indicates support, execute MONITOR inside a TRY/EXCEPT exception handler and trap for an exception. If an exception occurs, MONITOR and MWAIT are not supported at a privilege level greater than 0. See Example 8-24.

Example 8-24. Verifying MONITOR/MWAIT Support

```

boolean MONITOR_MWAIT_works = TRUE;
try {
    _asm {
        xor ecx, ecx
        xor edx, edx
        mov eax, MemArea
        monitor
    }
    // Use monitor
} except (UNWIND) {
    // if we get here, MONITOR/MWAIT is not supported
    MONITOR_MWAIT_works = FALSE;
}

```

8.10.4 MONITOR/MWAIT Instruction

Operating systems usually implement idle loops to handle thread synchronization. In a typical idle-loop scenario, there could be several “busy loops” and they would use a set of memory locations. An impacted processor waits in a loop and poll a memory location to determine if there is available work to execute. The posting of work is typically a write to memory (the work-queue of the waiting processor). The time for initiating a work request and getting it scheduled is on the order of a few bus cycles.

From a resource sharing perspective (logical processors sharing execution resources), use of the HLT instruction in an OS idle loop is desirable but has implications. Executing the HLT instruction on a idle logical processor puts the targeted processor in a non-execution state. This requires another processor (when posting work for the halted logical processor) to wake up the halted processor using an inter-processor interrupt. The posting and servicing of such an interrupt introduces a delay in the servicing of new work requests.

In a shared memory configuration, exits from busy loops usually occur because of a state change applicable to a specific memory location; such a change tends to be triggered by writes to the memory location by another agent (typically a processor).

MONITOR/MWAIT complement the use of HLT and PAUSE to allow for efficient partitioning and un-partitioning of shared resources among logical processors sharing physical resources. MONITOR sets up an effective address range that is monitored for write-to-memory activities; MWAIT places the processor in an optimized state (this may vary between different implementations) until a write to the monitored address range occurs.

In the initial implementation of MONITOR and MWAIT, they are available at CPL = 0 only.

Both instructions rely on the state of the processor’s monitor hardware. The monitor hardware can be either armed (by executing the MONITOR instruction) or triggered (due to a variety of events, including a store to the monitored memory region). If upon execution of MWAIT, monitor hardware is in a triggered state: MWAIT behaves as a NOP and execution continues at the next instruction in the execution stream. The state of monitor hardware is not architecturally visible except through the behavior of MWAIT.

Multiple events other than a write to the triggering address range can cause a processor that executed MWAIT to wake up. These include events that would lead to voluntary or involuntary context switches, such as:

- External interrupts, including NMI, SMI, INIT, BINIT, MCERR, A20M#
- Faults, Aborts (including Machine Check)
- Architectural TLB invalidations including writes to CR0, CR3, CR4 and certain MSR writes; execution of LMSW (occurring prior to issuing MWAIT but after setting the monitor)
- Voluntary transitions due to fast system call and far calls (occurring prior to issuing MWAIT but after setting the monitor)

Power management related events (such as Thermal Monitor 2 or chipset driven STPCLK# assertion) will not cause the monitor event pending flag to be cleared. Faults will not cause the monitor event pending flag to be cleared.

Software should not allow for voluntary context switches in between MONITOR/MWAIT in the instruction flow. Note that execution of MWAIT does not re-arm the monitor hardware. This means that MONITOR/MWAIT need to be executed in a loop. Also note that exits from the MWAIT state could be due to a condition other than a write to the triggering address; software should explicitly check the triggering data location to determine if the write occurred. Software should also check the value of the triggering address following the execution of the monitor instruction (and prior to the execution of the MWAIT instruction). This check is to identify any writes to the triggering address that occurred during the course of MONITOR execution.

The address range provided to the MONITOR instruction must be of write-back caching type. Only write-back memory type stores to the monitored address range will trigger the monitor hardware. If the address range is not in memory of write-back type, the address monitor hardware may not be set up properly or the monitor hardware may not be armed. Software is also responsible for ensuring that

- Writes that are not intended to cause the exit of a busy loop do not write to a location within the address region being monitored by the monitor hardware,
- Writes intended to cause the exit of a busy loop are written to locations within the monitored address region.

Not doing so will lead to more false wakeups (an exit from the MWAIT state not due to a write to the intended data location). These have negative performance implications. It might be necessary for software to use padding to prevent false wakeups. CPUID provides a mechanism for determining the size data locations for monitoring as well as a mechanism for determining the size of a the pad.

8.10.5 Monitor/Mwait Address Range Determination

To use the MONITOR/MWAIT instructions, software should know the length of the region monitored by the MONITOR/MWAIT instructions and the size of the coherence line size for cache-snoop traffic in a multiprocessor system. This information can be queried using the CPUID monitor leaf function (EAX = 05H). You will need the smallest and largest monitor line size:

- To avoid missed wake-ups: make sure that the data structure used to monitor writes fits within the smallest monitor line-size. Otherwise, the processor may not wake up after a write intended to trigger an exit from MWAIT.
- To avoid false wake-ups; use the largest monitor line size to pad the data structure used to monitor writes. Software must make sure that beyond the data structure, no unrelated data variable exists in the triggering area for MWAIT. A pad may be needed to avoid this situation.

These above two values bear no relationship to cache line size in the system and software should not make any assumptions to that effect. Within a single-cluster system, the two parameters should default to be the same (the size of the monitor triggering area is the same as the system coherence line size).

Based on the monitor line sizes returned by the CPUID, the OS should dynamically allocate structures with appropriate padding. If static data structures must be used by an OS, attempt to adapt the data structure and use a dynamically allocated data buffer for thread synchronization. When the latter technique is not possible, consider not using MONITOR/MWAIT when using static data structures.

To set up the data structure correctly for MONITOR/MWAIT on multi-clustered systems: interaction between processors, chipsets, and the BIOS is required (system coherence line size may depend on the chipset used in the system; the size could be different from the processor's monitor triggering area). The BIOS is responsible to set the correct value for system coherence line size using the IA32_MONITOR_FILTER_LINE_SIZE MSR. Depending on the relative magnitude of the size of the monitor triggering area versus the value written into the IA32_MONITOR_FILTER_LINE_SIZE MSR, the smaller of the parameters will be reported as the *Smallest Monitor Line Size*. The larger of the parameters will be reported as the *Largest Monitor Line Size*.

8.10.6 Required Operating System Support

This section describes changes that must be made to an operating system to run on processors supporting Intel Hyper-Threading Technology. It also describes optimizations that can help an operating system make more efficient use of the logical processors sharing execution resources. The required changes and suggested optimizations are representative of the types of modifications that appear in Windows* XP and Linux* kernel 2.4.0 operating systems for Intel processors supporting Intel Hyper-Threading Technology. Additional optimizations for processors

supporting Intel Hyper-Threading Technology are described in the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

8.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops

Intel recommends that a PAUSE instruction be placed in all spin-wait loops that run on Intel processors supporting Intel Hyper-Threading Technology and multi-core processors.

Software routines that use spin-wait loops include multiprocessor synchronization primitives (spin-locks, semaphores, and mutex variables) and idle loops. Such routines keep the processor core busy executing a load-compare-branch loop while a thread waits for a resource to become available. Including a PAUSE instruction in such a loop greatly improves efficiency (see Section 8.10.2, "PAUSE Instruction"). The following routine gives an example of a spin-wait loop that uses a PAUSE instruction:

```
Spin_Lock:
    CMP lockvar, 0    ;Check if lock is free
    JE Get_Lock
    PAUSE            ;Short delay
    JMP Spin_Lock

Get_Lock:
    MOV EAX, 1
    XCHG EAX, lockvar ;Try to get lock
    CMP EAX, 0      ;Test if successful
    JNE Spin_Lock

Critical_Section:
    <critical section code>
    MOV lockvar, 0
    ...
Continue:
```

The spin-wait loop above uses a "test, test-and-set" technique for determining the availability of the synchronization variable. This technique is recommended when writing spin-wait loops.

In IA-32 processor generations earlier than the Pentium 4 processor, the PAUSE instruction is treated as a NOP instruction.

8.10.6.2 Potential Usage of MONITOR/MWAIT in C0 Idle Loops

An operating system may implement different handlers for different idle states. A typical OS idle loop on an ACPI-compatible OS is shown in Example 8-25:

Example 8-25. A Typical OS Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1 handler
```

```

        // shown below
    }
}
// C1 handler uses a Halt instruction
VOID C1Handler()
{ STI
  HLT
}

```

The MONITOR and MWAIT instructions may be considered for use in the C0 idle state loops, if MONITOR and MWAIT are supported.

Example 8-26. An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop

```

// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

WHILE (1) {
  IF (WorkQueue) THEN {
    // Schedule work at WorkQueue.
  }
  ELSE {
    // No work to do - wait in appropriate C-state handler depending
    // on Idle time accumulated.
    IF (IdleTime >= IdleTimeThreshold) THEN {
      // Call appropriate C1, C2, C3 state handler, C1
      // handler shown below
      MONITOR WorkQueue // Setup of eax with WorkQueue
                        // LinearAddress,
                        // ECX, EDX = 0

      IF (WorkQueue = 0) THEN {
        MWAIT
      }
    }
  }
}
// C1 handler uses a Halt instruction.
VOID C1Handler()
{ STI
  HLT
}

```

8.10.6.3 Halt Idle Logical Processors

If one of two logical processors is idle or in a spin-wait loop of long duration, explicitly halt that processor by means of a HLT instruction.

In an MP system, operating systems can place idle processors into a loop that continuously checks the run queue for runnable software tasks. Logical processors that execute idle loops consume a significant amount of core's execution resources that might otherwise be used by the other logical processors in the physical package. For this reason, halting idle logical processors optimizes the performance.¹² If all logical processors within a physical package are halted, the processor will enter a power-saving state.

8.10.6.4 Potential Usage of MONITOR/MWAIT in C1 Idle Loops

An operating system may also consider replacing HLT with MONITOR/MWAIT in its C1 idle loop. An example is shown in Example 8-27:

Example 8-27. An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.
```

```
WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1
            // handler shown below
        }
    }
}

VOID C1Handler()

{ MONITOR WorkQueue // Setup of eax with WorkQueue LinearAddress,
    // ECX, EDX = 0
  IF (WorkQueue = 0) THEN {
    STI
    MWAIT // EAX, ECX = 0
  }
}
```

8.10.6.5 Guidelines for Scheduling Threads on Logical Processors Sharing Execution Resources

Because the logical processors, the order in which threads are dispatched to logical processors for execution can affect the overall efficiency of a system. The following guidelines are recommended for scheduling threads for execution.

- Dispatch threads to one logical processor per processor core before dispatching threads to the other logical processor sharing execution resources in the same processor core.
- In an MP system with two or more physical packages, distribute threads out over all the physical processors, rather than concentrate them in one or two physical processors.
- Use processor affinity to assign a thread to a specific processor core or package, depending on the cache-sharing topology. The practice increases the chance that the processor's caches will contain some of the thread's code and data when it is dispatched for execution after being suspended.

12. Excessive transitions into and out of the HALT state could also incur performance penalties. Operating systems should evaluate the performance trade-offs for their operating system.

8.10.6.6 Eliminate Execution-Based Timing Loops

Intel discourages the use of timing loops that depend on a processor's execution speed to measure time. There are several reasons:

- Timing loops cause problems when they are calibrated on a IA-32 processor running at one frequency and then executed on a processor running at another frequency.
- Routines for calibrating execution-based timing loops produce unpredictable results when run on an IA-32 processor supporting Intel Hyper-Threading Technology. This is due to the sharing of execution resources between the logical processors within a physical package.

To avoid the problems described, timing loop routines must use a timing mechanism for the loop that does not depend on the execution speed of the logical processors in the system. The following sources are generally available:

- A high resolution system timer (for example, an Intel 8254).
- A high resolution timer within the processor (such as, the local APIC timer or the time-stamp counter).

For additional information, see the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

8.10.6.7 Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory

When software uses locks or semaphores to synchronize processes, threads, or other code sections; Intel recommends that only one lock or semaphore be present within a cache line (or 128 byte sector, if 128-byte sector is supported). In processors based on Intel NetBurst microarchitecture (which support 128-byte sector consisting of two cache lines), following this recommendation means that each lock or semaphore should be contained in a 128-byte block of memory that begins on a 128-byte boundary. The practice minimizes the bus traffic required to service locks.

8.11 MP INITIALIZATION FOR P6 FAMILY PROCESSORS

This section describes the MP initialization process for systems that use multiple P6 family processors. This process uses the MP initialization protocol that was introduced with the Pentium Pro processor (see Section 8.4, "Multiple-Processor (MP) Initialization"). For P6 family processors, this protocol is typically used to boot 2 or 4 processors that reside on single system bus; however, it can support from 2 to 15 processors in a multi-clustered system when the APIC busses are tied together. Larger systems are not supported.

8.11.1 Overview of the MP Initialization Process For P6 Family Processors

During the execution of the MP initialization protocol, one processor is selected as the bootstrap processor (BSP) and the remaining processors are designated as application processors (APs), see Section 8.4.1, "BSP and AP Processors." Thereafter, the BSP manages the initialization of itself and the APs. This initialization includes executing BIOS initialization code and operating-system initialization code.

The MP protocol imposes the following requirements and restrictions on the system:

- An APIC clock (APICLK) must be provided.
- The MP protocol will be executed only after a power-up or RESET. If the MP protocol has been completed and a BSP has been chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each processor examines its BSP flag (in the APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

The following special-purpose interprocessor interrupts (IPIs) are used during the boot phase of the MP initialization protocol. These IPIs are broadcast on the APIC bus.

- Boot IPI (BIPI)—Initiates the arbitration mechanism that selects a BSP from the group of processors on the system bus and designates the remainder of the processors as APs. Each processor on the system bus broadcasts a BIPI to all the processors following a power-up or RESET.
- Final Boot IPI (FIPI)—Initiates the BIOS initialization procedure for the BSP. This IPI is broadcast to all the processors on the system bus, but only the BSP responds to it. The BSP responds by beginning execution of the BIOS initialization code at the reset vector.
- Startup IPI (SIPI)—Initiates the initialization procedure for an AP. The SIPI message contains a vector to the AP initialization code in the BIOS.

Table 8-5 describes the various fields of the boot phase IPIs.

Table 8-5. Boot Phase IPI Message Format

Type	Destination Field	Destination Shorthand	Trigger Mode	Level	Destination Mode	Delivery Mode	Vector (Hex)
BIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	40 to 4E*
FIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	10
SIPI	Used	All excluding self	Edge	Assert	Physical	StartUp (110)	00 to FF

NOTE:

* For all P6 family processors.

For BIPI messages, the lower 4 bits of the vector field contain the APIC ID of the processor issuing the message and the upper 4 bits contain the "generation ID" of the message. All P6 family processor will have a generation ID of 4H. BIPIs will therefore use vector values ranging from 40H to 4EH (4FH can not be used because FH is not a valid APIC ID).

8.11.2 MP Initialization Protocol Algorithm

Following a power-up or RESET of a system, the P6 family processors in the system execute the MP initialization protocol algorithm to initialize each of the processors on the system bus. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each processor on the system bus is assigned a unique APIC ID, based on system topology (see Section 8.4.5, "Identifying Logical Processors in an MP System"). This ID is written into the local APIC ID register for each processor.
2. Each processor executes its internal BIST simultaneously with the other processors on the system bus. Upon completion of the BIST (at T0), each processor broadcasts a BIPI to "all including self" (see Figure 8-1).
3. APIC arbitration hardware causes all the APICs to respond to the BIPIs one at a time (at T1, T2, T3, and T4).
4. When the first BIPI is received (at time T1), each APIC compares the four least significant bits of the BIPI's vector field with its APIC ID. If the vector and APIC ID match, the processor selects itself as the BSP by setting the BSP flag in its IA32_APIC_BASE MSR. If the vector and APIC ID do not match, the processor selects itself as an AP by entering the "wait for SIPI" state. (Note that in Figure 8-1, the BIPI from processor 1 is the first BIPI to be handled, so processor 1 becomes the BSP.)
5. The newly established BSP broadcasts an FIPI message to "all including self." The FIPI is guaranteed to be handled only after the completion of the BIPIs that were issued by the non-BSP processors.

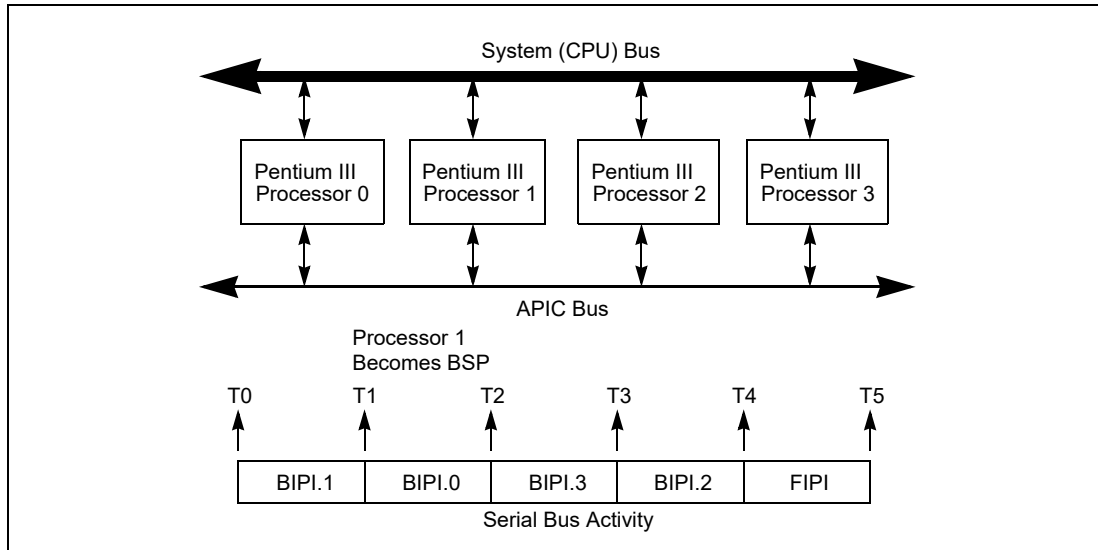


Figure 8-1. MP System With Multiple Pentium III Processors

6. After the BSP has been established, the outstanding BIPIs are received one at a time (at T2, T3, and T4) and ignored by all processors.
7. When the FIPI is finally received (at T5), only the BSP responds to it. It responds by fetching and executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
8. As part of the boot-strap code, the BSP creates an ACPI table and an MP table and adds its initial APIC ID to these tables as appropriate.
9. At the end of the boot-strap procedure, the BSP broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000V V000H, where VV is the vector contained in the SIPI message).
10. All APs respond to the SIPI message by racing to a BIOS initialization semaphore. The first one to the semaphore begins executing the initialization code. (See MP init code for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and MP tables as appropriate. At the completion of the initialization procedure, the AP executes a CLI instruction (to clear the IF flag in the EFLAGS register) and halts itself.
11. When each of the APs has gained access to the semaphore and executed the AP initialization code and all written their APIC IDs into the appropriate places in the ACPI and MP tables, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
12. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

See Section 8.4.4, "MP Initialization Example," for an annotated example the use of the MP protocol to boot IA-32 processors in an MP. This code should run on any IA-32 processor that used the MP protocol.

8.11.2.1 Error Detection and Handling During the MP Initialization Protocol

Errors may occur on the APIC bus during the MP initialization phase. These errors may be transient or permanent and can be caused by a variety of failure mechanisms (for example, broken traces, soft errors during bus usage, etc.). All serial bus related errors will result in an APIC checksum or acceptance error.

The MP initialization protocol makes the following assumptions regarding errors that occur during initialization:

- If errors are detected on the APIC bus during execution of the MP initialization protocol, the processors that detect the errors are shut down.
- The MP initialization protocol will be executed by processors even if they fail their BIST sequences.

18. Updates to Chapter 9, Volume 3A

Change bars and green text show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Added details on 5-level paging where applicable. Typo corrections as needed.

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

9.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor's registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 8.4, "Multiple-Processor (MP) Initialization," for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.
- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).
- **Pentium processors** — In either a single- or dual- processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.
- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.

9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor in real-address mode with paging disabled.

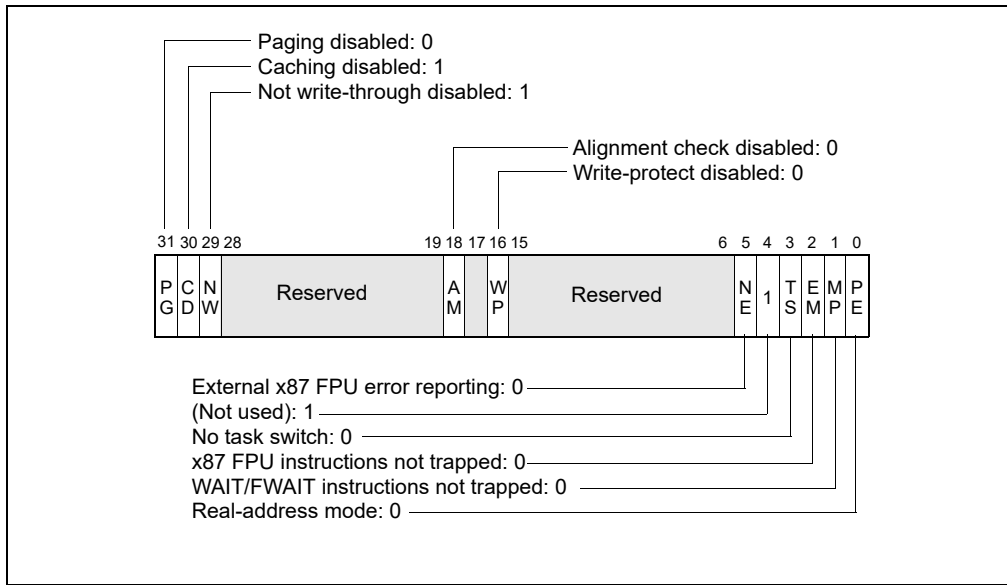


Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS ¹	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H ²	60000010H ²	60000010H ²
CR2, CR3, CR4	00000000H	00000000H	00000000H
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	000n06xxH ³	000n06xxH ³	000n06xxH ³
EAX	0 ⁴	0 ⁴	0 ⁴
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7 ⁵	+0.0	+0.0	FINIT/FNINIT: Unchanged

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
x87 FPU Control Word ⁵	0040H	0040H	FINIT/FNINIT: 037FH
x87 FPU Status Word ⁵	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Tag Word ⁵	5555H	5555H	FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors ⁵	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers ⁵	00000000H	00000000H	FINIT/FNINIT: 00000000H
MM0 through MM7 ⁵	0000000000000000H	0000000000000000H	INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	0H	0H	Unchanged
MXCSR	1F80H	1F80H	Unchanged
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
DR7	00000400H	00000400H	00000400H
R8-R15	0000000000000000H	0000000000000000H	0000000000000000H
XMM8-XMM15	0H	0H	Unchanged
XCRO	1H	1H	Unchanged
IA32_XSS	0H	0H	0H
YMM_H[255:128]	0H	0H	Unchanged
BNDCFGU	0H	0H	0H
BND0-BND3	0H	0H	0H
IA32_BNDCFGS	0H	0H	0H
OPMASK	0H	0H	Unchanged
ZMM_H[511:256]	0H	0H	Unchanged
ZMMHi16[511:0]	0H	0H	Unchanged
PKRU	0H	0H	Unchanged
Intel Processor Trace MSRs	0H	0H ^w	Unchanged
Time-Stamp Counter	0H	0H ^w	Unchanged
IA32_TSC_AUX	0H	0H	Unchanged
IA32_TSC_ADJUST	0H	0H	Unchanged
IA32_TSC_DEADLINE	0H	0H	Unchanged
IA32_SYSENTER_CS/ESP/EIP	0H	0H	Unchanged
IA32_EFER	0000000000000000H	0000000000000000H	0000000000000000H
IA32_STAR/LSTAR	0H	0H	Unchanged
IA32_FS_BASE/GS_BASE	0H	0H	0H

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
IA32_PMCx, IA32_PERFEVTSELx	0H	0H	Unchanged
IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL, Global Perf Counter Controls	0H	0H	Unchanged
Data and Code Cache, TLBs	Invalid ⁶	Invalid ⁶	Unchanged
Fixed MTRRs	Disabled	Disabled	Unchanged
Variable MTRRs	Disabled	Disabled	Unchanged
Machine-Check Banks	Undefined	Undefined ^w	Unchanged
Last Branch Record Stack	0	0 ^w	Unchanged
APIC	Enabled	Enabled	Unchanged
X2APIC	Disabled	Disabled	Unchanged
IA32_DEBUG_INTERFACE	0	0 ^w	Unchanged

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, and “xx” = don’t care.
 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.
- W: Warm RESET behavior differs from power-on RESET with details listed in Table 9-2.

Table 9-2. Variance of RESET Values in Selected Intel Architecture Processors

State	XREF	Value	Feature Flag or DisplayFamily_DisplayModel Signatures
Time-Stamp Counter	Warm RESET	Unmodified across warm Reset	06_2DH, 06_3EH
Machine-Check Banks	Warm RESET	IA32_MCi_Status banks are unmodified across warm Reset	06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H
Last Branch Record Stack	Warm RESET	LBR stack MSRs are unmodified across warm Reset	06_1AH, 06_1CH, DisplayFamiy= 06 and DisplayModel > 1DH
Intel Processor Trace MSRs	Warm RESET	Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified	If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1
IA32_DEBUG_INTERFACE	Warm RESET	Unmodified across warm Reset	If CPUID.01H:ECX.[1 1] = 1

9.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

9.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 9-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

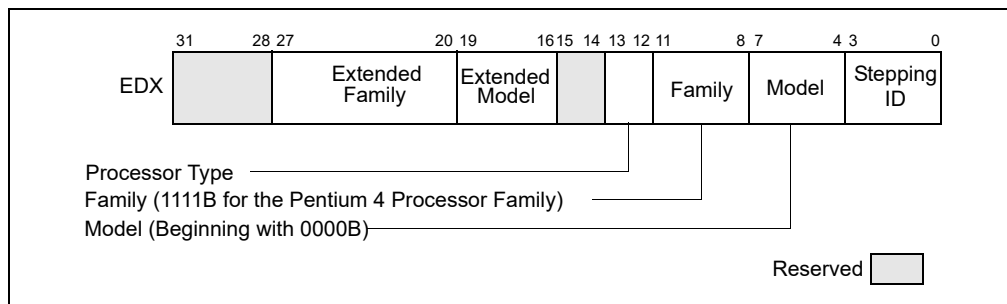


Figure 9-2. Version Information in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

9.2 X87 FPU INITIALIZATION

Software-initialization code can determine the whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.

A hardware reset places the x87 FPU in the state shown in Table 9-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 9-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

9.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 9-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

Table 9-3. Recommended Settings of EM and MP Flags on IA-32 Processors

EM	MP	NE	IA-32 processor
1	0	1	Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.
0	1	1 or 0*	Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.
0	1	1 or 0*	More recent Intel 64 or IA-32 processors

NOTE:

* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

9.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 9-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.

To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 9-4.

Table 9-4. Software Emulation Settings of EM, MP, and NE Flags

CRO Bit	Value
EM	1
MP	0
NE	1

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

9.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor's caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 11, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

9.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 23, "Introduction to Virtual Machine Extensions").
- Debug extensions (see Chapter 23, "Introduction to Virtual Machine Extensions").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 15, "Machine-Check Architecture").
- MTRRs (see Section 11.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32_EFER, IA32_FEATURE_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.

Lists of available performance-monitoring events are given in Chapter 19, “Performance Monitoring Events”, and lists of available MSRs are given in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

9.5 MEMORY TYPE RANGE REGISTERS (MTRRS)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 11.11, “Memory Type Range Registers (MTRRs),” for detailed information on the MTRRs.

9.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support for the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.
2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, “Control Registers,” for a description of the OSFXSR flag.
3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, “Control Registers,” for a description of the OSXMMEXCPT flag.
4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See “MXCSR Control and Status Register” in Chapter 10, “Programming with Streaming SIMD Extensions (SSE),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a detailed description of the bits and flags in the MXCSR register.

9.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 9.8, “Software Initialization for Protected-Mode Operation.”

9.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the “interrupt vector table”). By default, the address of the base of the IDT is physical address 0H. This address can be

changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

9.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

- A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

9.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 9.7, "Software Initialization for Real-Address Mode Operation." Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

- A IDT.
- A GDT.
- A TSS.
- (Optional) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

- The GDTR.
- (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
- Control registers CR1 through CR4.
- (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

9.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.

To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 9.8.3, "Initializing Paging."

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

9.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

9.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

9.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary to load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.

After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

9.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four or five levels of enhanced paging structures (see Section 4.5, “4-Level Paging and 5-Level Paging”).

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4) or Level 5 page map table (PML5).
4. Enable IA-32e mode by setting IA32_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging structures must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS on an attempt to activate IA-32e mode.

9.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.

9.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

9.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

9.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

9.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

9.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 9.8, “Software Initialization for Protected-Mode Operation.” Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To insure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
 - Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
 - Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
 - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
 - Insure that the GDT and IDT are in identity mapped pages.
 - Clear the PG bit in the CR0 register.
 - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
 - Limit = 64 KBytes (0FFFFH)
 - Byte granular (G = 0)
 - Expand up (E = 0)
 - Writable (W = 1)
 - Present (P = 1)
 - Base = any value

The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.

5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

9.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.

- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 9-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor’s physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 9-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 9-1. The line numbers given in Table 9-5 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.

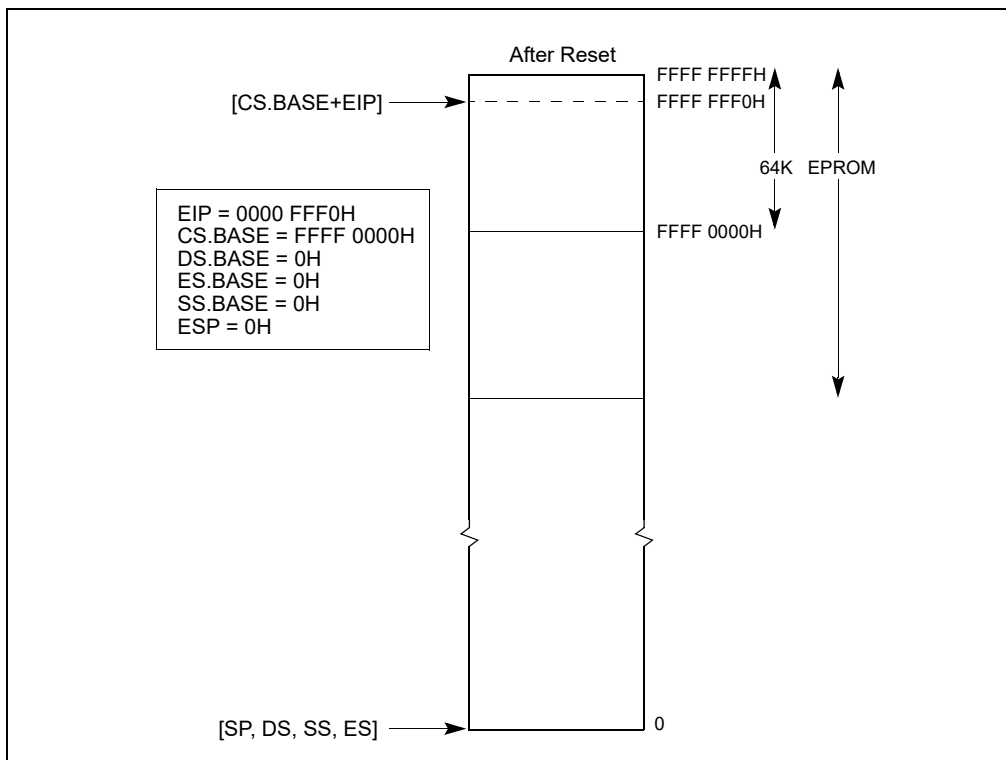


Figure 9-3. Processor State After Reset

Table 9-5. Main Initialization Steps in STARTUP.ASM Source Listing

STARTUP.ASM Line Numbers		Description
From	To	
157	157	Jump (short) to the entry code in the EPROM
162	169	Construct a temporary GDT in RAM with one entry: 0 - null 1 - R/W data segment, base = 0, limit = 4 GBytes
171	172	Load the GDTR to point to the temporary GDT
174	177	Load CRO with PE flag set to switch to protected mode
179	181	Jump near to clear real mode instruction queue
184	186	Load DS, ES registers with GDT[1] descriptor, so both point to the entire physical memory space
188	195	Perform specific board initialization that is imposed by the new protected mode
196	218	Copy the application's GDT from ROM into RAM
220	238	Copy the application's IDT from ROM into RAM
241	243	Load application's GDTR
244	245	Load application's IDTR
247	261	Copy the application's TSS from ROM into RAM
263	267	Update TSS descriptor and other aliases in GDT (GDT alias or IDT alias)
277	277	Load the task register (without task switch) using LTR instruction
282	286	Load SS, ESP with the value found in the application's TSS
287	287	Push EFLAGS value found in the application's TSS
288	288	Push CS value found in the application's TSS
289	289	Push EIP value found in the application's TSS
290	293	Load DS, ES with the value found in the application's TSS
296	296	Perform IRET; pop the above values and enter the application code

9.10.1 Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.
- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.
- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default- segment attribute will be used to generate the right opcode.

9.10.2 STARTUP.ASM Listing

Example 9-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.

Example 9-1. STARTUP.ASM

MS-DOS* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP 09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP
 OBJECT MODULE PLACED IN startup.obj
 ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132)

```

LINE      SOURCE

1         NAME      STARTUP
2
3         ;;;;;;;;;;;;;;
4         ;
5         ;   ASSUMPTIONS:
6         ;
7         ;       1.  The bottom 64K of memory is ram, and can be used for
8         ;           scratch space by this module.
9         ;
10        ;       2.  The system has sufficient free usable ram to copy the
11        ;           initial GDT, IDT, and TSS
12        ;
13        ;;;;;;;;;;;;;;
14
15        ; configuration data - must match with build definition
16
17        CS_BASE      EQU      0FFFF0000H
18
19        ; CS_BASE is the linear address of the segment STARTUP_CODE
20        ; - this is specified in the build language file
21
22        RAM_START     EQU      400H
23
24        ; RAM_START is the start of free, usable ram in the linear
25        ; memory space.  The GDT, IDT, and initial TSS will be
26        ; copied above this space, and a small data segment will be
27        ; discarded at this linear address.  The 32-bit word at
28        ; RAM_START will contain the linear address of the first
29        ; free byte above the copied tables - this may be useful if
30        ; a memory manager is used.
31
32        TSS_INDEX     EQU      10
33
34        ; TSS_INDEX is the index of the TSS of the first task to
35        ; run after startup
36
37
38        ;;;;;;;;;;;;;;
39
40        ; ----- STRUCTURES and EQU -----
41        ; structures for system data
42
43        ; TSS structure
44        TASK_STATE    STRUC
45        link          DW ?

```

PROCESSOR MANAGEMENT AND INITIALIZATION

```
46     link_h     DW ?
47     ESP0      DD ?
48     SS0       DW ?
49     SS0_h     DW ?
50     ESP1      DD ?
51     SS1       DW ?
52     SS1_h     DW ?
53     ESP2      DD ?
54     SS2       DW ?
55     SS2_h     DW ?
56     CR3_reg   DD ?
57     EIP_reg   DD ?
58     EFLAGS_reg DD ?
59     EAX_reg   DD ?
60     ECX_reg   DD ?
61     EDX_reg   DD ?
62     EBX_reg   DD ?
63     ESP_reg   DD ?
64     EBP_reg   DD ?
65     ESI_reg   DD ?
66     EDI_reg   DD ?
67     ES_reg    DW ?
68     ES_h     DW ?
69     CS_reg    DW ?
70     CS_h     DW ?
71     SS_reg    DW ?
72     SS_h     DW ?
73     DS_reg    DW ?
74     DS_h     DW ?
75     FS_reg    DW ?
76     FS_h     DW ?
77     GS_reg    DW ?
78     GS_h     DW ?
79     LDT_reg   DW ?
80     LDT_h     DW ?
81     TRAP_reg  DW ?
82     IO_map_base DW ?
83     TASK_STATE ENDS
84
85     ; basic structure of a descriptor
86     DESC      STRUC
87         lim_0_15 DW ?
88         bas_0_15 DW ?
89         bas_16_23 DB ?
90         access   DB ?
91         gran     DB ?
92         bas_24_31 DB ?
93     DESC      ENDS
94
95     ; structure for use with LGDT and LIDT instructions
96     TABLE_REG STRUC
97         table_lim DW ?
98         table_linear DD ?
99     TABLE_REG ENDS
```

```

100
101 ; offset of GDT and IDT descriptors in builder generated GDT
102 GDT_DESC_OFF EQU 1*SIZE(DESC)
103 IDT_DESC_OFF EQU 2*SIZE(DESC)
104
105 ; equates for building temporary GDT in RAM
106 LINEAR_SEL EQU 1*SIZE(DESC)
107 LINEAR_PROTO_LO EQU 00000FFFFH ; LINEAR_ALIAS
108 LINEAR_PROTO_HI EQU 000CF9200H
109
110 ; Protection Enable Bit in CR0
111 PE_BIT EQU 1B
112
113 ; -----
114
115 ; ----- DATA SEGMENT-----
116
117 ; Initially, this data segment starts at linear 0, according
118 ; to the processor's power-up state.
119
120 STARTUP_DATA SEGMENT RW
121
122 free_mem_linear_base LABEL DWORD
123 TEMP_GDT LABEL BYTE ; must be first in segment
124 TEMP_GDT_NULL_DESC DESC <>
125 TEMP_GDT_LINEAR_DESC DESC <>
126
127 ; scratch areas for LGDT and LIDT instructions
128 TEMP_GDT_SCRATCH TABLE_REG <>
129 APP_GDT_RAM TABLE_REG <>
130 APP_IDT_RAM TABLE_REG <>
131 ; align end_data
132 fill DW ?
133
134 ; last thing in this segment - should be on a dword boundary
135 end_data LABEL BYTE
136
137 STARTUP_DATA ENDS
138 ; -----
139
140
141 ; ----- CODE SEGMENT-----
142 STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144 ; filled in by builder
145 PUBLIC GDT_EPROM
146 GDT_EPROM TABLE_REG <>
147
148 ; filled in by builder
149 PUBLIC IDT_EPROM
150 IDT_EPROM TABLE_REG <>
151
152 ; entry point into startup code - the bootstrap will vector
153 ; here with a near JMP generated by the builder. This

```

PROCESSOR MANAGEMENT AND INITIALIZATION

```
154 ; label must be in the top 64K of linear memory.
155
156     PUBLIC  STARTUP
157 STARTUP:
158
159 ; DS,ES address the bottom 64K of flat linear memory
160     ASSUME  DS:STARTUP_DATA, ES:STARTUP_DATA
161 ; See Figure 9-4
162 ; load GDTR with temporary GDT
163     LEA    EBX,TEMP_GDT ; build the TEMP_GDT in low ram,
164     MOV    DWORD PTR [EBX],0 ; where we can address
165     MOV    DWORD PTR [EBX]+4,0
166     MOV    DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167     MOV    DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168     MOV    TEMP_GDT_scratch.table_linear,EBX
169     MOV    TEMP_GDT_scratch.table_lim,15
170
171     DB 66H; execute a 32 bit LGDT
172     LGDT  TEMP_GDT_scratch
173
174 ; enter protected mode
175     MOV    EBX,CR0
176     OR    EBX,PE_BIT
177     MOV    CR0,EBX
178
179 ; clear prefetch queue
180     JMP    CLEAR_LABEL
181 CLEAR_LABEL:
182
183 ; make DS and ES address 4G of linear memory
184     MOV    CX,LINEAR_SEL
185     MOV    DS,CX
186     MOV    ES,CX
187
188 ; do board specific initialization
189 ;
190 ;
191 ; .....
192 ;
193
194
195 ; See Figure 9-5
196 ; copy EPROM GDT to ram at:
197 ;          RAM_START + size (STARTUP_DATA)
198     MOV    EAX,RAM_START
199     ADD    EAX,OFFSET (end_data)
200     MOV    EBX,RAM_START
201     MOV    ECX, CS_BASE
202     ADD    ECX, OFFSET (GDT_EPROM)
203     MOV    ESI, [ECX].table_linear
204     MOV    EDI,EAX
205     MOVZX ECX, [ECX].table_lim
206     MOV    APP_GDT_ram[EBX].table_lim,CX
```

```

207     INC     ECX
208     MOV     EDX,EAX
209     MOV     APP_GDT_ram[EBX].table_linear,EAX
210     ADD     EAX,ECX
211     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213     ; fixup GDT base in descriptor
214     MOV     ECX,EDX
215     MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216     ROR     ECX,16
217     MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218     MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220     ; copy EPROM IDT to ram at:
221     ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222     MOV     ECX, CS_BASE
223     ADD     ECX, OFFSET (IDT_EPROM)
224     MOV     ESI, [ECX].table_linear
225     MOV     EDI,EAX
226     MOVZX  ECX, [ECX].table_lim
227     MOV     APP_IDT_ram[EBX].table_lim,CX
228     INC     ECX
229     MOV     APP_IDT_ram[EBX].table_linear,EAX
230     MOV     EBX,EAX
231     ADD     EAX,ECX
232     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234     ; fixup IDT pointer in GDT
235     MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236     ROR     EBX,16
237     MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238     MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240     ; load GDTR and IDTR
241     MOV     EBX,RAM_START
242     DB     66H           ; execute a 32 bit LGDT
243     LGDT   APP_GDT_ram[EBX]
244     DB     66H           ; execute a 32 bit LIDT
245     LIDT   APP_IDT_ram[EBX]
246
247     ; move the TSS
248     MOV     EDI,EAX
249     MOV     EBX,TSS_INDEX*SIZE(DESC)
250     MOV     ECX,GDT_DESC_OFF ;build linear address for TSS
251     MOV     GS,CX
252     MOV     DH,GS:[EBX].bas_24_31
253     MOV     DL,GS:[EBX].bas_16_23
254     ROL     EDX,16
255     MOV     DX,GS:[EBX].bas_0_15
256     MOV     ESI,EDX
257     LSL     ECX,EBX
258     INC     ECX
259     MOV     EDX,EAX
260     ADD     EAX,ECX

```

PROCESSOR MANAGEMENT AND INITIALIZATION

```
261     REP MOVS     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
262
263         ; fixup TSS pointer
264     MOV     GS:[EBX].bas_0_15,DX
265     ROL     EDX,16
266     MOV     GS:[EBX].bas_24_31,DH
267     MOV     GS:[EBX].bas_16_23,DL
268     ROL     EDX,16
269     ;save start of free ram at linear location RAMSTART
270     MOV     free_mem_linear_base+RAM_START,EAX
271
272     ;assume no LDT used in the initial task - if necessary,
273     ;code to move the LDT could be added, and should resemble
274     ;that used to move the TSS
275
276     ; load task register
277     LTR     BX ; No task switch, only descriptor loading
278     ; See Figure 9-6
279     ; load minimal set of registers necessary to simulate task
280     ; switch
281
282
283     MOV     AX,[EDX].SS_reg ; start loading registers
284     MOV     EDI,[EDX].ESP_reg
285     MOV     SS,AX
286     MOV     ESP,EDI ; stack now valid
287     PUSH   DWORD PTR [EDX].EFLAGS_reg
288     PUSH   DWORD PTR [EDX].CS_reg
289     PUSH   DWORD PTR [EDX].EIP_reg
290     MOV     AX,[EDX].DS_reg
291     MOV     BX,[EDX].ES_reg
292     MOV     DS,AX ; DS and ES no longer linear memory
293     MOV     ES,BX
294
295     ; simulate far jump to initial task
296     IRETD
297
298     STARTUP_CODE ENDS
*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
299
300     END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA
301
302
```

ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.

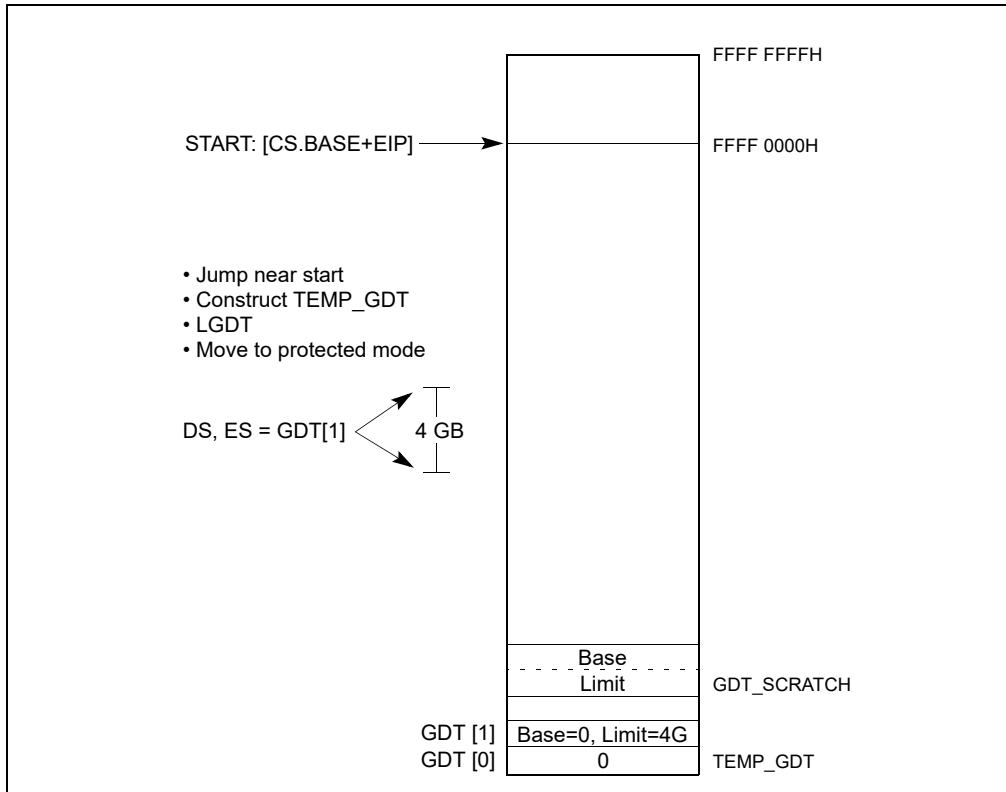


Figure 9-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

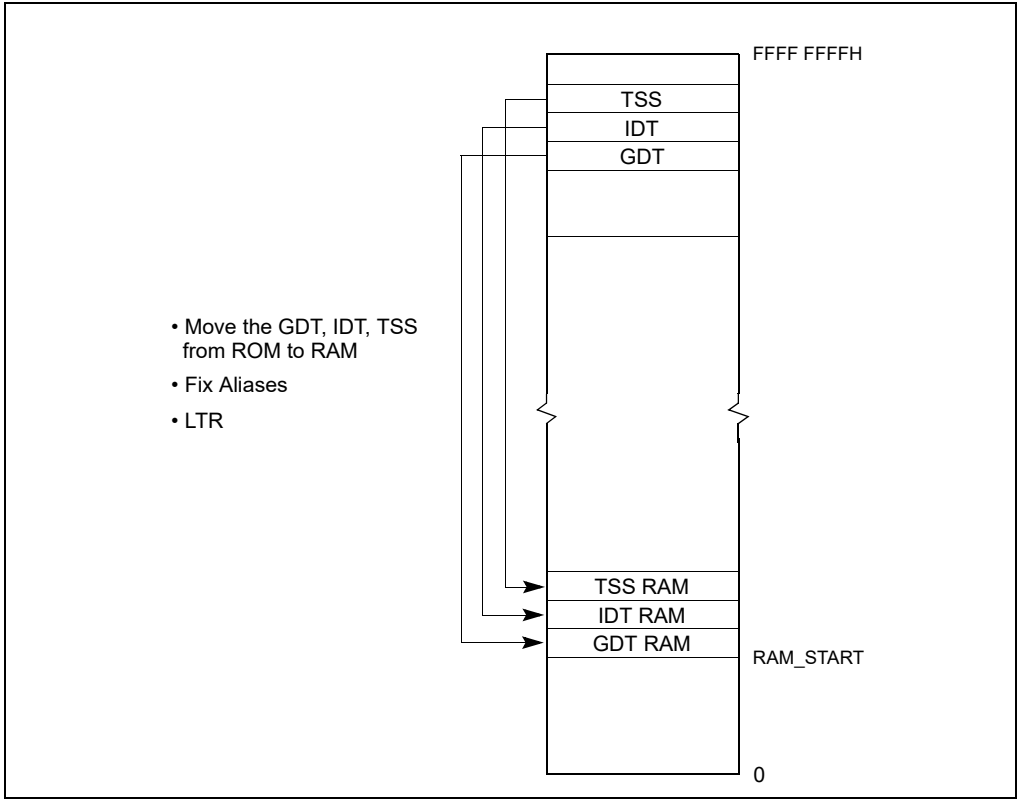


Figure 9-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

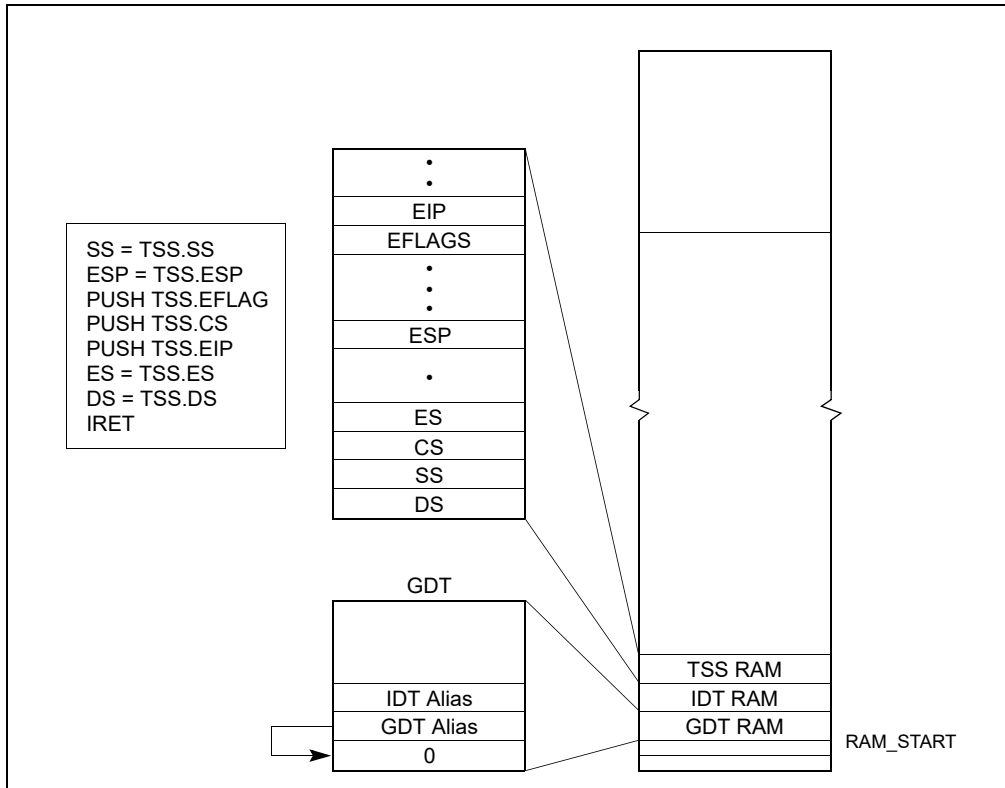


Figure 9-6. Task Switching (Lines 282-296 of List File)

9.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 9-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

Example 9-2. MAIN.ASM

```

NAME    main_module
data    SEGMENT RW
        dw 1000 dup(?)
DATA    ENDS
stack  stackseg 800
CODE SEGMENT ER use32 PUBLIC
main_start:
        nop
        nop
        nop
CODE    ENDS
END main_start, ds:data, ss:stack

```

9.10.4 Supporting Files

The batch file shown in Example 9-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.

Example 9-3. Batch File to Assemble and Build the Application

```
ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload
```

BLD386 performs several operations in this example:
 It allocates physical memory location to segments and tables.
 It generates tables using the build file and the input files.
 It links object files and resolves references.
 It generates a boot-loadable file to be programmed into the EPROM.

Example 9-4 shows the build file used as an input to BLD386 to perform the above functions.

Example 9-4. Build File

```
INIT_BLD_EXAMPLE;

SEGMENT
    *SEGMENTS(DPL = 0)
    , startup.startup_code(BASE = 0FFFF0000H)
    ;

TASK
    BOOT_TASK(OBJECT = startup, INITIAL,DPL = 0,
              NOT INTENABLED)
    , PROTECTED_MODE_TASK(OBJECT = main_module,DPL = 0,
                          NOT INTENABLED)
    ;

TABLE
    GDT (
        LOCATION = GDT_EPROM
        , ENTRY = (
            10: PROTECTED_MODE_TASK
            , startup.startup_code
            , startup.startup_data
            , main_module.data
            , main_module.code
            , main_module.stack
            )
        ),
    IDT (
        LOCATION = IDT_EPROM
        );

MEMORY
    (
        RESERVE = (0..3FFFH
                  -- Area for the GDT, IDT, TSS copied from ROM
                  , 60000H..0FFFEFFFFH)
        , RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFHH)
                  -- Eprom size 64K
                  , RANGE = (RAM_AREA = RAM (4000H..05FFFFH))
```

);

END

Table 9-6 shows the relationship of each build item with an ASM source file.

Table 9-6. Relationship Between BLD Item and ASM Source File

Item	ASM386 and Startup.A58	BLD386 Controls and BLD file	Effect
Bootstrap	public startup startup:	bootstrap start(startup)	Near jump at OFFFFFFFF0H to start.
GDT location	public GDT_EPROM GDT_EPROM TABLE_REG <>	TABLE GDT(location = GDT_EPROM)	The location of the GDT will be programmed into the GDT_EPROM location.
IDT location	public IDT_EPROM IDT_EPROM TABLE_REG <>	TABLE IDT(location = IDT_EPROM)	The location of the IDT will be programmed into the IDT_EPROM location.
RAM start	RAM_START equ 400H	memory (reserve = (0..3FFFH))	RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.
Location of the application TSS in the GDT	TSS_INDEX EQU 10	TABLE GDT(ENTRY = (10: PROTECTED_MODE_ TASK))	Put the descriptor of the application TSS in GDT entry 10.
EPROM size and location	size and location of the initialization code	SEGMENT startup.code (base = OFFF0000H) ...memory (RANGE(ROM_AREA = ROM(x..y))	Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.

9.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

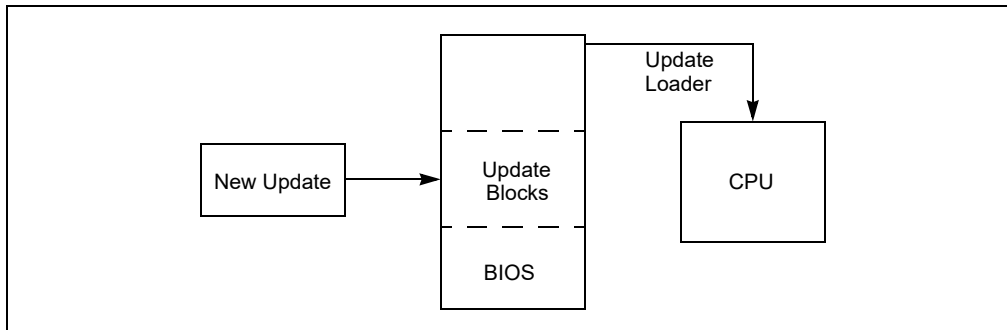


Figure 9-7. Applying Microcode Updates

9.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor’s signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0fH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2 for details).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-7 provides a definition of the fields; Table 9-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

Table 9-7. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature	12	4	<i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model, and stepping</i> of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.
Reserved	36	12	Reserved fields for future expansion.
Update Data	48	Data Size or 2000	Update data.
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields.

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature[n]	Data Size + 68 + (n * 12)	4	<p><i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model</i>, and <i>stepping</i> of the processor.</p> <p>Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.</p>
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	<p>Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table.</p> <p>To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.</p>

Table 9-8. Microcode Update Format

31	24	16	8	0	Bytes		
Header Version					0		
Update Revision					4		
Month: 8		Day: 8		Year: 16	8		
Processor Signature (CPUID)					12		
Res: 4	Extended Family: 8	Extended Mode: 4	Reserved: 2	Type: 2	Family: 4	Model: 4	Stepping: 4
Checksum					16		
Loader Revision					20		
Processor Flags					24		
Reserved (24 bits)					P7 P6 P5 P4 P3 P2 P1 P0		
Data Size					28		
Total Size					32		
Reserved (12 Bytes)					36		

Table 9-8. Microcode Update Format (Contd.)

31	24	16	8	0	Bytes
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)					48
Extended Signature Count 'n'					Data Size + 48
Extended Processor Signature Table Checksum					Data Size + 52
Reserved (12 Bytes)					Data Size + 56
Processor Signature[n]					Data Size + 68 + (n * 12)
Processor Flags[n]					Data Size + 72 + (n * 12)
Checksum[n]					Data Size + 76 + (n * 12)

9.11.2 Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

Table 9-9. Extended Processor Signature Table Header Structure

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

Table 9-10. Processor Signature Structure

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.3 Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 9-5 shows how to check for a valid processor signature match between the processor and microcode update.

Example 9-5. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
    // first check the ProcessorSignature field
    If (ProcessorSignature = Update.ProcessorSignature)
        Success

    // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].
        //

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
            (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++);

            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
            If (N < Update.ExtendedSignatureCount)
                Success
            Else
                Fail
    }
    Else
        Fail
Else
    Fail
```

9.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: IA32_PLATFORM_ID
MSR Address: 017H

Access: Read Only

IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

Table 9-11. Processor Flags

Bit	Descriptions																																				
63:53	Reserved																																				
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-8. <table style="margin-left: 40px; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;">52</td> <td style="padding-right: 10px;">51</td> <td style="padding-right: 10px;">50</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>Processor Flag 0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>Processor Flag 1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>Processor Flag 2</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>Processor Flag 3</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>Processor Flag 4</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>Processor Flag 5</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>Processor Flag 6</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>Processor Flag 7</td> </tr> </table>	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
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																																		
49:0	Reserved																																				

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-6.

Example 9-6. Pseudo Code Example of Processor Flags Test

```

Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
  If (Update.ProcessorFlags & Flag)
  {
    Load Update
  }
  Else
  {

    //
    // Assume the Data Size has been used to calculate the
    // location of Update.ProcessorSignature[N] and a match
    // on Update.ProcessorSignature[N] has already succeeded
    //

    If (Update.ProcessorFlags[n] & Flag)
    {
      Load Update
    }
  }
}

```

9.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum

procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = $(Total\ Size / 4)$, where the total size is a multiple of 1024 bytes (1 KBytes).

Example 9-7. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize ≠ 00000000H)
    N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
    ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum = 00000000H)
    Success
Else
    Fail
```

9.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

Example 9-8. Assembly Code Example of Simple Microcode Update Loader

```
mov  ecx,79h           ; MSR to write in ECX
xor  eax,eax          ; clear EAX
xor  ebx,ebx          ; clear EBX
mov  ax,cs             ; Segment of microcode update
shl  eax,4
mov  bx,offset Update ; Offset of microcode update
add  eax,ebx           ; Linear Address of Update in EAX
add  eax,48d          ; Offset of the Update Data within the Update
xor  edx,edx           ; Zero in EDX
WRMSR                  ; microcode update trigger
```

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:

- The addresses for the microcode update data must be in canonical form.
- If paging is enabled, the microcode update data must map that data as present.
- The microcode update data must start at a 16-byte aligned linear address.

9.11.6.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

9.11.6.2 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

9.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology

Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

9.11.6.4 Update in a System Supporting Dual-Core Technology

Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 9.11.6.3 also applies.

9.11.6.5 Update Loader Enhancements

The update loader presented in Section 9.11.6, "Microcode Update Loader," is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.

- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7, “Update Signature and Verification.”
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5, “Microcode Update Checksum.”
- A loader can provide power-on messages indicating successful loading of an update.

9.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different stepings.

9.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-9.

Example 9-9. Assembly Code to Retrieve the Update Revision

```

MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
XOR    EAX, EAX           ;clear EAX
XOR    EDX, EDX           ;clear EDX
WRMSR                ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
rdmsr                ;Read Model Specific Register
    
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	Microcode Update Signature Register
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-12).

Table 9-12. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

9.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-10.

Example 9-10. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
  Load Update that is to be authenticated;
  Y ← Obtain New Signature from MSR 8BH;

  If (Z = Y)
    Success
  Else
    Fail
}
Else
  Fail
```

Example 9-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

9.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g. similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 9.11.8.1-9.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g. an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification (www.uefi.org). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

9.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 9-11. Pseudo Code, Checks Required Prior to Loading an Update

```

For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
        {

```

```

If ((Update.ProcessorSignature = Processor Signature) &&
    (Update.ProcessorFlags & Platform Bits))
{
    Load Update.UpdateData into the Processor;
    Verify update was correctly loaded into the processor
    Go on to next processor
    Break;
}
Else If (Update.TotalSize > (Update.DataSize + 48))
{
    N ← 0
    While (N < Update.ExtendedSignatureCount)
    {
        If ((Update.ProcessorSignature[N] =
            Processor Signature) &&
            (Update.ProcessorFlags[N] & Platform Bits))
        {
            Load Update.UpdateData into the Processor;
            Verify update correctly loaded into the processor
            Go on to next processor
            Break;
        }
        N ← N + 1
    }
    I ← I + (Update.TotalSize / 2048)
    If ((Update.TotalSize MOD 2048) = 0)
        I ← I + 1
    }
}
}
}
}

```

NOTES

The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

9.11.8.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an

older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-12 represents a calling program.

Example 9-12. INT 15 D042 Calling Program Pseudo-code

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e., Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
    Id Bits

    If current processor cannot be updated
        exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}
//
// Do we have enough update slots for all CPUs?
//
```

```

If there are more blocks required to support the unique processor steppings than update blocks
provided by the BIOS exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks = 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
    {
        Display Error -- BIOS is not managing NVRAM appropriately
        exit
    }

    If (INVALID_REVISION) returned
    {
        Display Message: More recent update already loaded in NVRAM for
        this stepping
        continue
    }

    If any other error returned
    {
        Display Diagnostic
        exit
    }

    //
    // Verify the update was loaded correctly
    //
    Issue the ReadUpdate function

    If an error occurred
    {
        Display Diagnostic
        exit
    }
    //
    // Compare the Update read to that written
    //
    If (Update read ≠ Update written)
    {
        Display Diagnostic
        exit
    }

    I ← I + (size of microcode update / 2048)
}
//
// Enable Update Loading, and inform user

```

//
Issue the Update Control function with Task = Enable.

9.11.8.3 Microcode Update Functions

Table 9-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

Table 9-13. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

9.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

9.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-14 lists the parameters and return codes for the function.

Table 9-14. Parameters for the Presence Test

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP' - Part two of the signature
EDX	Loader Version	Version number of the microcode update loader

Table 9-14. Parameters for the Presence Test (Contd.)

Input		
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

9.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-15 lists the parameters and return codes for the function.

Table 9-15. Parameters for the Write Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or... Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 KBytes of RAM block
SS:SP	Stack pointer	32 KBytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

Table 9-15. Parameters for the Write Update Data Function (Contd.)

Input	
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	The processor stepping does not currently exist in the system.
INVALID_HEADER	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The update does not checksum correctly.
SECURITY_FAILURE	The processor rejected the update.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6, "Microcode Update Loader." This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

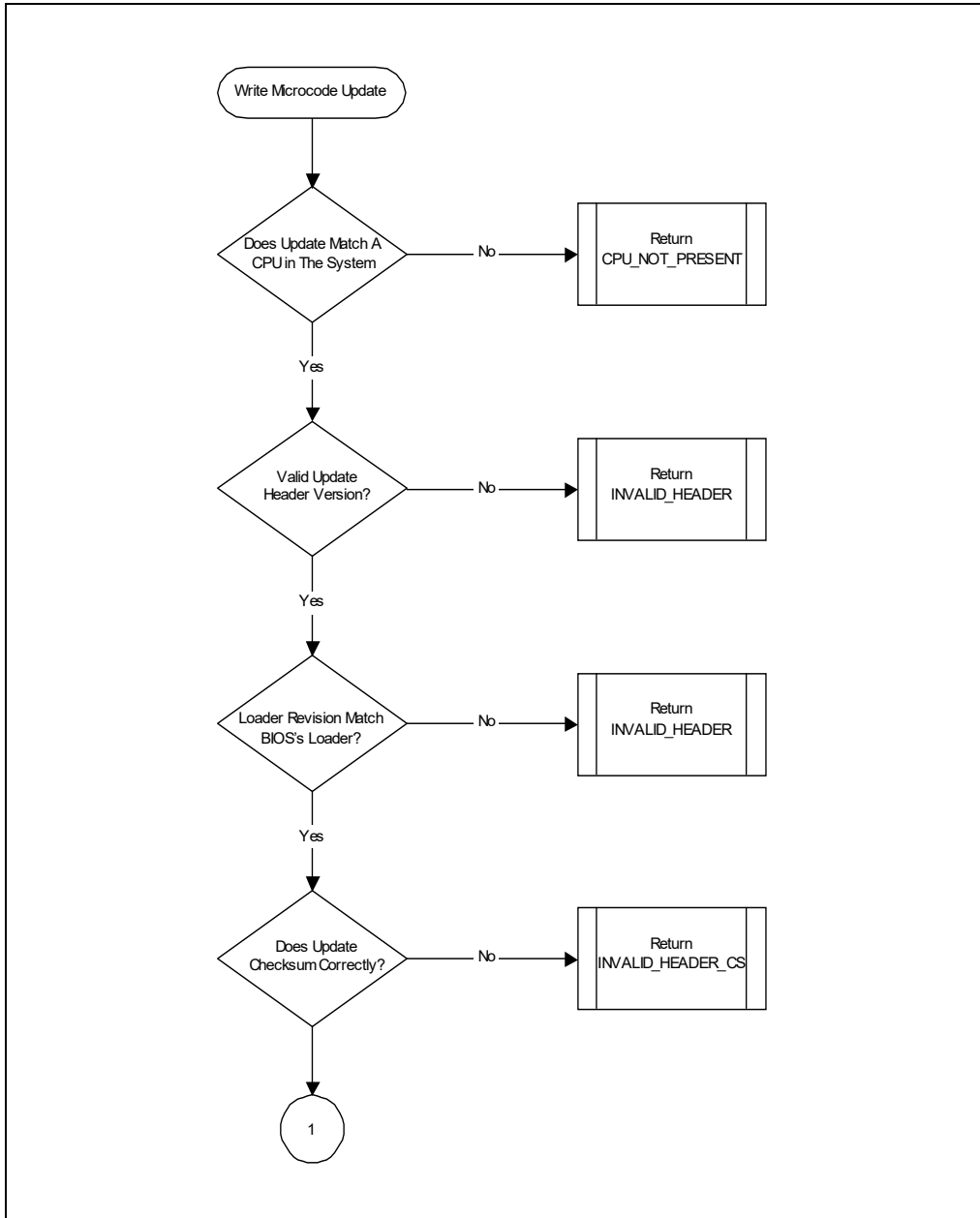


Figure 9-8. Microcode Update Write Operation Flow [1]

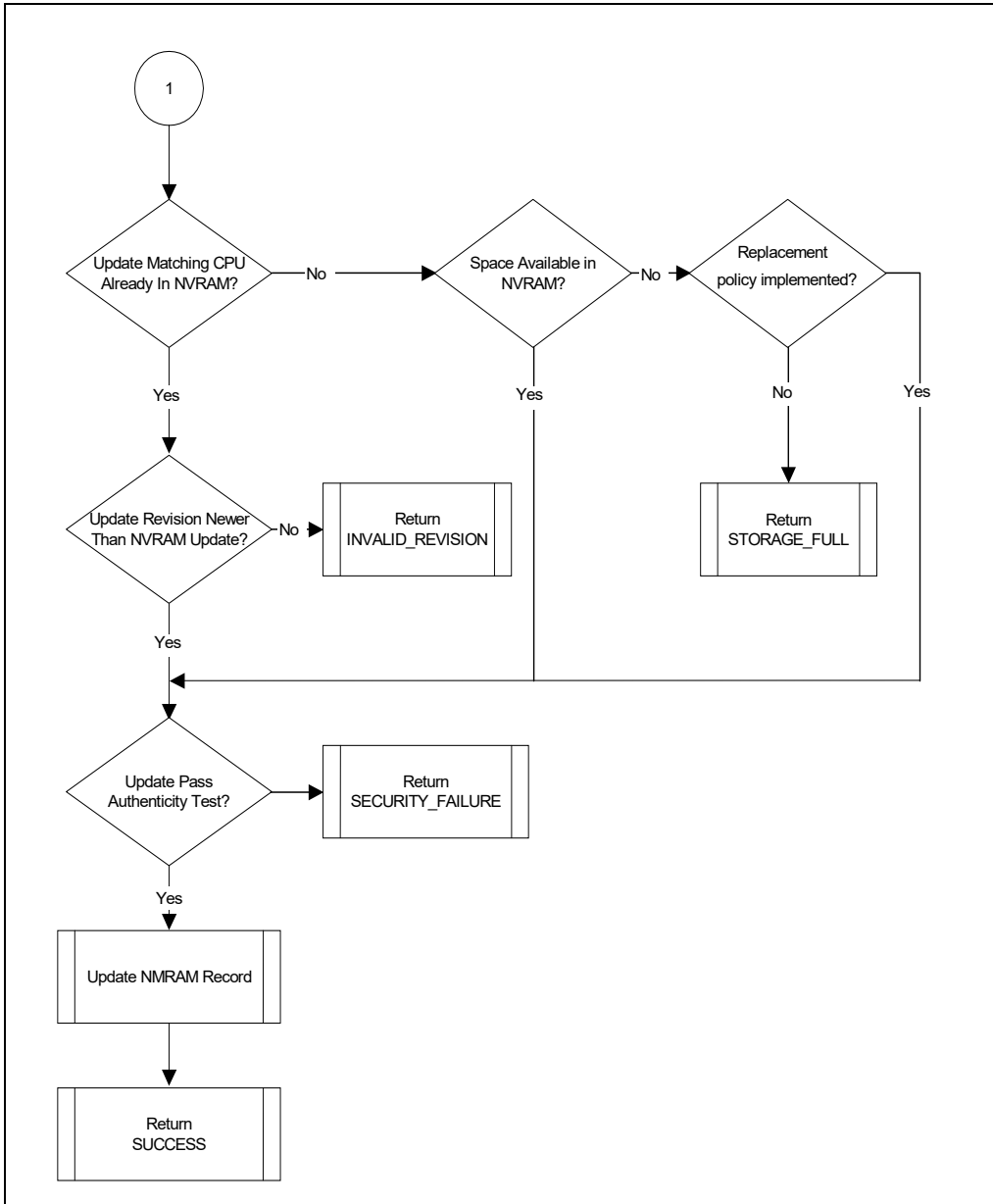


Figure 9-9. Microcode Update Write Operation Flow [2]

9.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-16 lists the parameters and return codes for the function.

Table 9-16. Parameters for the Control Update Sub-function

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 KBytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

Table 9-17. Mnemonic Values

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

9.11.8.8 Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-18 lists the parameters and return codes.

Table 9-18. Parameters for the Read Microcode Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data

Table 9-18. Parameters for the Read Microcode Update Data Function (Contd.)

ECX	Scratch Pad1	Real Mode Segment address of 64 KBytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 KBytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 KBytes of RAM Block
SS:SP	Stack pointer	32 KBytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
Output		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFH after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

9.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 9-19 are available in the AH register.

Table 9-19. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented.
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

19. Updates to Chapter 10, Volume 3A

Change bars and green text show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Typo corrections as needed.

CHAPTER 10

ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in the following sections as the local APIC, was introduced into the IA-32 processors with the Pentium processor (see Section 22.27, “Advanced Programmable Interrupt Controller (APIC)”) and is included in the P6 family, Pentium 4, Intel Xeon processors, and other more recent Intel 64 and IA-32 processor families (see Section 10.4.2, “Presence of the Local APIC”). The local APIC performs two primary functions for the processor:

- It receives interrupts from the processor’s interrupt pins, from internal sources and from an external I/O APIC (or other external interrupt controller). It sends these to the processor core for handling.
- In multiple processor (MP) systems, it sends and receives interprocessor interrupt (IPI) messages to and from other logical processors on the system bus. IPI messages can be used to distribute interrupts among the processors in the system or to execute system wide functions (such as, booting up processors or distributing work among a group of processors).

The external **I/O APIC** is part of Intel’s system chip set. Its primary function is to receive external interrupt events from the system and its associated I/O devices and relay them to the local APIC as interrupt messages. In MP systems, the I/O APIC also provides a mechanism for distributing external interrupts to the local APICs of selected processors or groups of processors on the system bus.

This chapter provides a description of the local APIC and its programming interface. It also provides an overview of the interface between the local APIC and the I/O APIC. Contact Intel for detailed information about the I/O APIC.

When a local APIC has sent an interrupt to its processor core for handling, the processor uses the interrupt and exception handling mechanism described in Chapter 6, “Interrupt and Exception Handling.” See Section 6.1, “Interrupt and Exception Overview,” for an introduction to interrupt and exception handling.

10.1 LOCAL AND I/O APIC OVERVIEW

Each local APIC consists of a set of APIC registers (see Table 10-1) and associated hardware that control the delivery of interrupts to the processor core and the generation of IPI messages. The APIC registers are memory mapped and can be read and written to using the MOV instruction.

Local APICs can receive interrupts from the following sources:

- **Locally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected directly to the processor’s local interrupt pins (LINT0 and LINT1). The I/O devices may also be connected to an 8259-type interrupt controller that is in turn connected to the processor through one of the local interrupt pins.
- **Externally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected to the interrupt input pins of an I/O APIC. Interrupts are sent as I/O interrupt messages from the I/O APIC to one or more of the processors in the system.
- **Inter-processor interrupts (IPIs)** — An Intel 64 or IA-32 processor can use the IPI mechanism to interrupt another processor or group of processors on the system bus. IPIs are used for software self-interrupts, interrupt forwarding, or preemptive scheduling.
- **APIC timer generated interrupts** — The local APIC timer can be programmed to send a local interrupt to its associated processor when a programmed count is reached (see Section 10.5.4, “APIC Timer”).
- **Performance monitoring counter interrupts** — P6 family, Pentium 4, and Intel Xeon processors provide the ability to send an interrupt to its associated processor when a performance-monitoring counter overflows (see Section 18.6.3.5.8, “Generating an Interrupt on Overflow”).
- **Thermal Sensor interrupts** — Pentium 4 and Intel Xeon processors provide the ability to send an interrupt to themselves when the internal thermal sensor has been tripped (see Section 14.8.2, “Thermal Monitor”).

- APIC internal error interrupts** — When an error condition is recognized within the local APIC (such as an attempt to access an unimplemented register), the APIC can be programmed to send an interrupt to its associated processor (see Section 10.5.3, "Error Handling").

Of these interrupt sources: the processor’s LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector are referred to as **local interrupt sources**. Upon receiving a signal from a local interrupt source, the local APIC delivers the interrupt to the processor core using an interrupt delivery protocol that has been set up through a group of APIC registers called the **local vector table** or **LVT** (see Section 10.5.1, "Local Vector Table"). A separate entry is provided in the local vector table for each local interrupt source, which allows a specific interrupt delivery protocol to be set up for each source. For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt) to the processor core.

The local APIC handles interrupts from the other two interrupt sources (externally connected I/O devices and IPIs) through its IPI message handling facilities.

A processor can generate IPIs by programming the interrupt command register (ICR) in its local APIC (see Section 10.6.1, "Interrupt Command Register (ICR)"). The act of writing to the ICR causes an IPI message to be generated and issued on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for Pentium and P6 family processors). See Section 10.2, "System Bus Vs. APIC Bus."

IPIs can be sent to other processors in the system or to the originating processor (self-interrupts). When the target processor receives an IPI message, its local APIC handles the message automatically (using information included in the message such as vector number and trigger mode). See Section 10.6, "Issuing Interprocessor Interrupts," for a detailed explanation of the local APIC’s IPI message delivery and acceptance mechanism.

The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 10-1). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as interrupt messages.

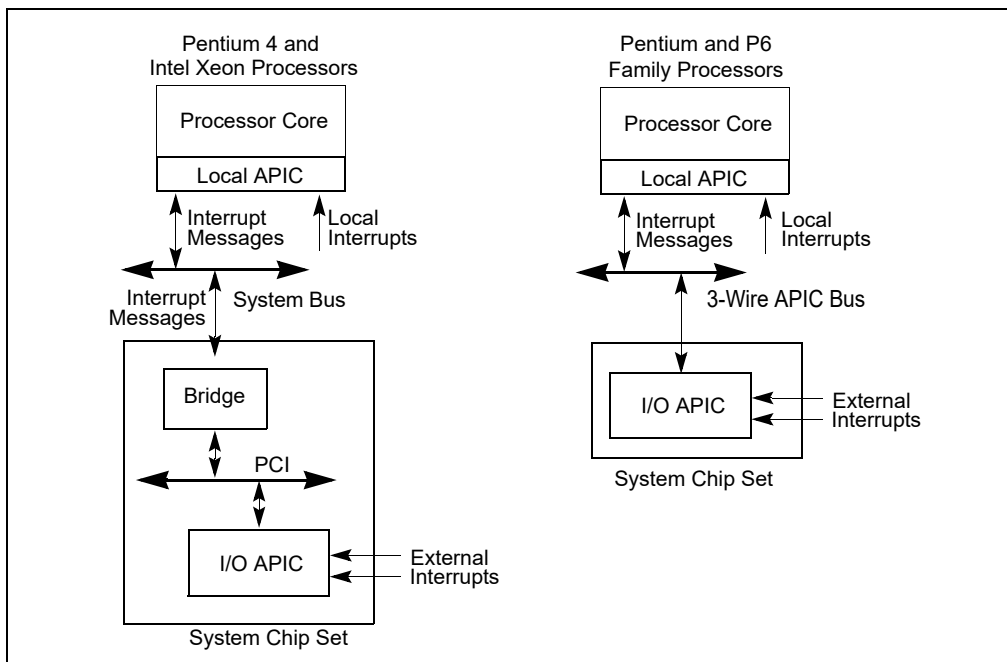


Figure 10-1. Relationship of Local APIC and I/O APIC In Single-Processor Systems

Individual pins on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. The I/O APIC also has a "virtual wire mode" that allows it to communicate with a standard 8259A-style external interrupt controller. Note that the local APIC can be disabled (see Section 10.4.3, "Enabling or Disabling the Local APIC"). This allows an associated processor core to receive interrupts directly from an 8259A interrupt controller.

Both the local APIC and the I/O APIC are designed to operate in MP systems (see Figures 10-2 and 10-3). Each local APIC handles interrupts from the I/O APIC, IPIs from processors on the system bus, and self-generated interrupts. Interrupts can also be delivered to the individual processors through the local interrupt pins; however, this mechanism is commonly not used in MP systems.

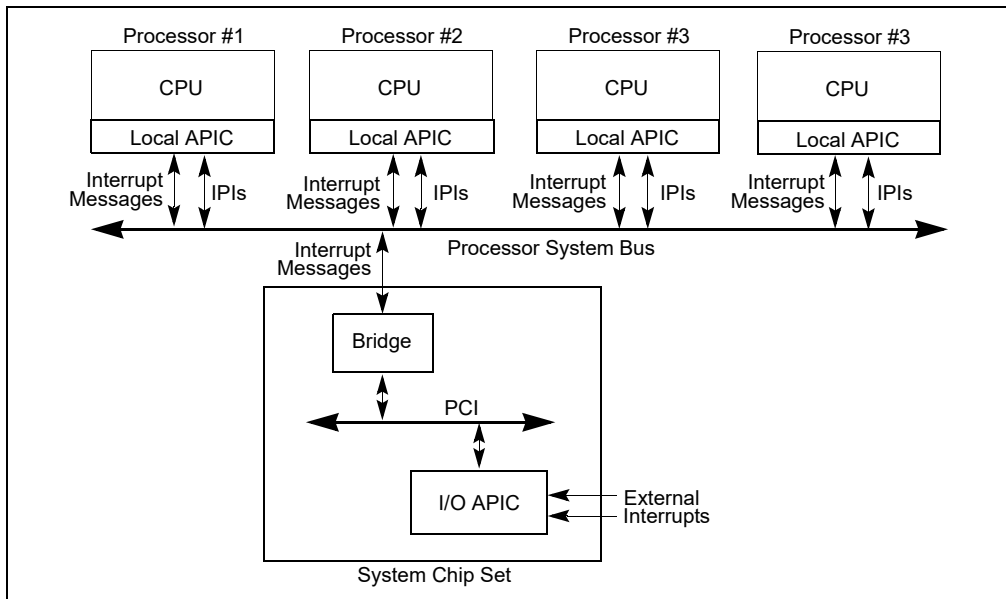


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

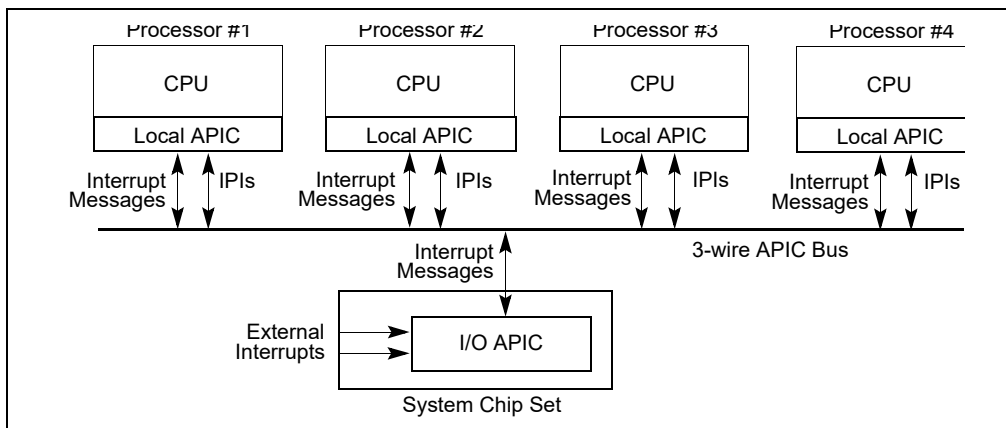


Figure 10-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems

The IPI mechanism is typically used in MP systems to send fixed interrupts (interrupts for a specific vector number) and special-purpose interrupts to processors on the system bus. For example, a local APIC can use an IPI to forward a fixed interrupt to another processor for servicing. Special-purpose IPIs (including NMI, INIT, SMI and SIPI IPIs) allow one or more processors on the system bus to perform system-wide boot-up and control functions.

The following sections focus on the local APIC and its implementation in the Pentium 4, Intel Xeon, and P6 family processors. In these sections, the terms "local APIC" and "I/O APIC" refer to local and I/O APICs used with the P6 family processors and to local and I/O xAPICs used with the Pentium 4 and Intel Xeon processors (see Section 10.3, "The Intel® 82489DX External APIC, the APIC, the xAPIC, and the X2APIC").

10.2 SYSTEM BUS VS. APIC BUS

For the P6 family and Pentium processors, the I/O APIC and local APICs communicate through the 3-wire inter-APIC bus (see Figure 10-3). Local APICs also use the APIC bus to send and receive IPIs. The APIC bus and its messages are invisible to software and are not classed as architectural.

Beginning with the Pentium 4 and Intel Xeon processors, the I/O APIC and local APICs (using the xAPIC architecture) communicate through the system bus (see Figure 10-2). The I/O APIC sends interrupt requests to the processors on the system bus through bridge hardware that is part of the Intel chip set. The bridge hardware generates the interrupt messages that go to the local APICs. IPIs between local APICs are transmitted directly on the system bus.

10.3 THE INTEL® 82489DX EXTERNAL APIC, THE APIC, THE XAPIC, AND THE X2APIC

The local APIC in the P6 family and Pentium processors is an architectural subset of the Intel® 82489DX external APIC. See Section 22.27.1, “Software Visible Differences Between the Local APIC and the 82489DX.”

The APIC architecture used in the Pentium 4 and Intel Xeon processors (called the xAPIC architecture) is an extension of the APIC architecture found in the P6 family processors. The primary difference between the APIC and xAPIC architectures is that with the xAPIC architecture, the local APICs and the I/O APIC communicate through the system bus. With the APIC architecture, they communicate through the APIC bus (see Section 10.2, “System Bus Vs. APIC Bus”). Also, some APIC architectural features have been extended and/or modified in the xAPIC architecture. These extensions and modifications are described in Section 10.4 through Section 10.10.

The basic operating mode of the xAPIC is **xAPIC mode**. The x2APIC architecture is an extension of the xAPIC architecture, primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. These extensions and modifications are supported by a new mode of execution (**x2APIC mode**) are detailed in Section 10.12.

10.4 LOCAL APIC

The following sections describe the architecture of the local APIC and how to detect it, identify it, and determine its status. Descriptions of how to program the local APIC are given in Section 10.5.1, “Local Vector Table,” and Section 10.6.1, “Interrupt Command Register (ICR).”

10.4.1 The Local APIC Block Diagram

Figure 10-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor’s physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 11.3, “Methods of Caching Available.”

In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 10.4.5, “Relocating the Local APIC Registers,” describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation both in xAPIC mode and (if enabled by software) in x2APIC mode. x2APIC mode provides extended processor addressability (see Section 10.12).

NOTE

For P6 family, Pentium 4, and Intel Xeon processors, the APIC handles all memory accesses to addresses within the 4-KByte APIC register space internally and no external bus cycles are produced. For the Pentium processors with an on-chip APIC, bus cycles are produced for accesses to the APIC register space. Thus, for software intended to run on Pentium processors, system software should explicitly not map the APIC register space to regular system memory. Doing so can result in an invalid opcode exception (#UD) being generated or unpredictable execution.

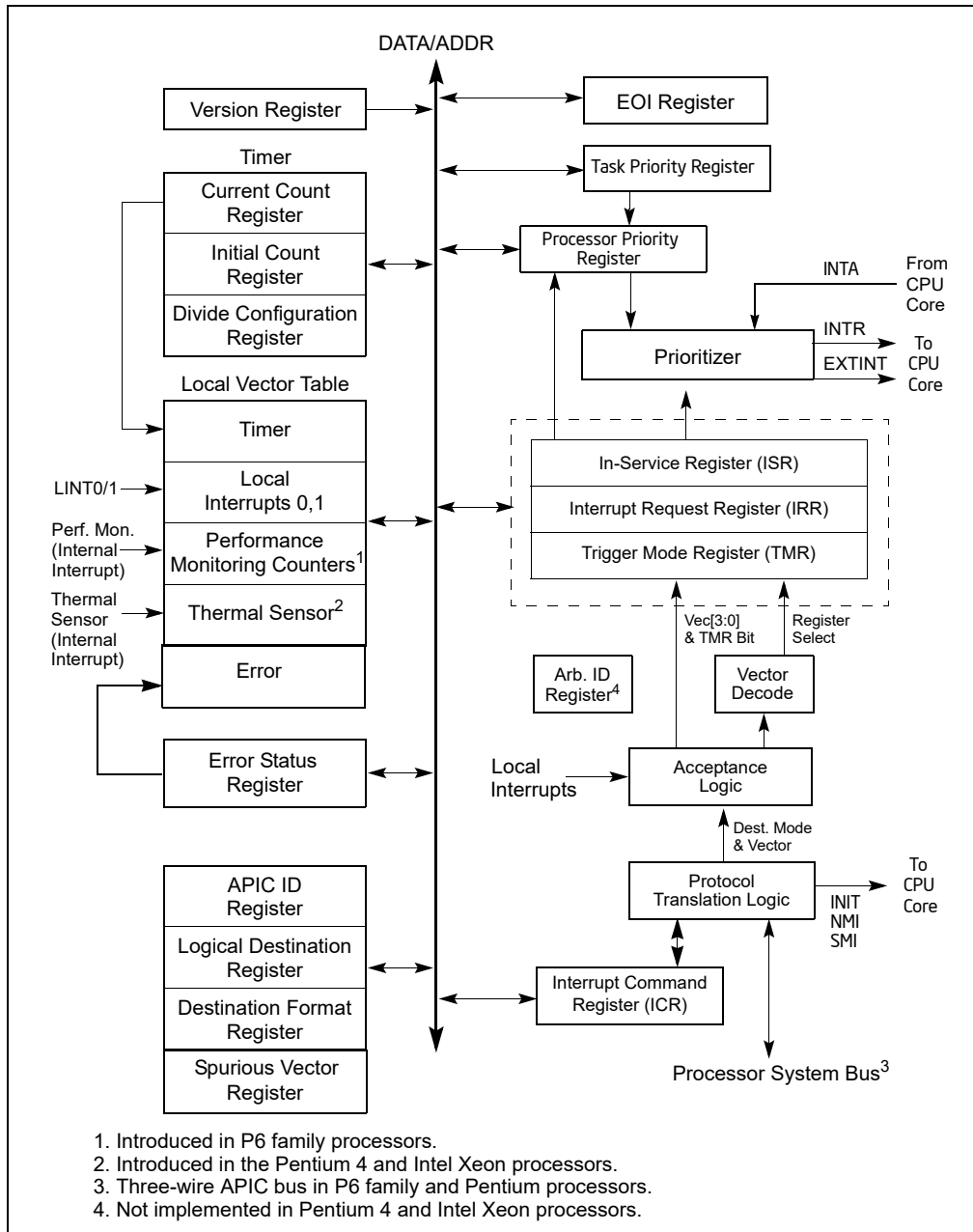


Figure 10-4. Local APIC Structure

Table 10-1 shows how the APIC registers are mapped into the 4-KByte APIC register space. Registers are 32 bits, 64 bits, or 256 bits in width; all are aligned on 128-bit boundaries. All 32-bit registers should be accessed using 128-bit aligned 32-bit loads or stores. Some processors may support loads and stores of less than 32 bits to some of the APIC registers. This is model specific behavior and is not guaranteed to work on all processors. Any

FP/MMX/SSE access to an APIC register, or any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results or unexpected exceptions, including machine checks, and may vary between implementations. Wider registers (64-bit or 256-bit) must be accessed using multiple 32-bit loads or stores, with all accesses being 128-bit aligned. The local APIC registers listed in Table 10-1 are not MSRs. The only MSR associated with the programming of the local APIC is the IA32_APIC_BASE MSR (see Section 10.4.3, “Enabling or Disabling the Local APIC”).

NOTE

In processors based on Intel microarchitecture code name Nehalem¹ the Local APIC ID Register is no longer Read/Write; it is Read Only.

Table 10-1 Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register ¹ (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Remote Read Register ¹ (RRD)	Read Only
FEE0 00D0H	Logical Destination Register	Read/Write.
FEE0 00E0H	Destination Format Register	Read/Write (see Section 10.6.2.2).
FEE0 00F0H	Spurious Interrupt Vector Register	Read/Write (see Section 10.9.
FEE0 0100H	In-Service Register (ISR); bits 31:0	Read Only.
FEE0 0110H	In-Service Register (ISR); bits 63:32	Read Only.
FEE0 0120H	In-Service Register (ISR); bits 95:64	Read Only.
FEE0 0130H	In-Service Register (ISR); bits 127:96	Read Only.
FEE0 0140H	In-Service Register (ISR); bits 159:128	Read Only.
FEE0 0150H	In-Service Register (ISR); bits 191:160	Read Only.
FEE0 0160H	In-Service Register (ISR); bits 223:192	Read Only.
FEE0 0170H	In-Service Register (ISR); bits 255:224	Read Only.
FEE0 0180H	Trigger Mode Register (TMR); bits 31:0	Read Only.
FEE0 0190H	Trigger Mode Register (TMR); bits 63:32	Read Only.
FEE0 01A0H	Trigger Mode Register (TMR); bits 95:64	Read Only.

1. See Table 2-1, “CPUID Signature Values of DisplayFamily_DisplayModel,” on page 1, and Section 2.8, “MSRs In the Intel® Microarchitecture Code Name Nehalem” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* to determine which processors are based on Nehalem microarchitecture.

Table 10-1 Local APIC Register Address Map (Contd.)

Address	Register Name	Software Read/Write
FEE0 01B0H	Trigger Mode Register (TMR); bits 127:96	Read Only.
FEE0 01C0H	Trigger Mode Register (TMR); bits 159:128	Read Only.
FEE0 01D0H	Trigger Mode Register (TMR); bits 191:160	Read Only.
FEE0 01E0H	Trigger Mode Register (TMR); bits 223:192	Read Only.
FEE0 01F0H	Trigger Mode Register (TMR); bits 255:224	Read Only.
FEE0 0200H	Interrupt Request Register (IRR); bits 31:0	Read Only.
FEE0 0210H	Interrupt Request Register (IRR); bits 63:32	Read Only.
FEE0 0220H	Interrupt Request Register (IRR); bits 95:64	Read Only.
FEE0 0230H	Interrupt Request Register (IRR); bits 127:96	Read Only.
FEE0 0240H	Interrupt Request Register (IRR); bits 159:128	Read Only.
FEE0 0250H	Interrupt Request Register (IRR); bits 191:160	Read Only.
FEE0 0260H	Interrupt Request Register (IRR); bits 223:192	Read Only.
FEE0 0270H	Interrupt Request Register (IRR); bits 255:224	Read Only.
FEE0 0280H	Error Status Register	Read Only.
FEE0 0290H through FEE0 02E0H	Reserved	
FEE0 02F0H	LVT Corrected Machine Check Interrupt (CMCI) Register	Read/Write.
FEE0 0300H	Interrupt Command Register (ICR); bits 0-31	Read/Write.
FEE0 0310H	Interrupt Command Register (ICR); bits 32-63	Read/Write.
FEE0 0320H	LVT Timer Register	Read/Write.
FEE0 0330H	LVT Thermal Sensor Register ²	Read/Write.
FEE0 0340H	LVT Performance Monitoring Counters Register ³	Read/Write.
FEE0 0350H	LVT LINT0 Register	Read/Write.
FEE0 0360H	LVT LINT1 Register	Read/Write.
FEE0 0370H	LVT Error Register	Read/Write.
FEE0 0380H	Initial Count Register (for Timer)	Read/Write.
FEE0 0390H	Current Count Register (for Timer)	Read Only.
FEE0 03A0H through FEE0 03D0H	Reserved	
FEE0 03E0H	Divide Configuration Register (for Timer)	Read/Write.
FEE0 03F0H	Reserved	

NOTES:

1. Not supported in the Pentium 4 and Intel Xeon processors. The Illegal Register Access bit (7) of the ESR will not be set when writing to these registers.
2. Introduced in the Pentium 4 and Intel Xeon processors. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.
3. Introduced in the Pentium Pro processor. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.

10.4.2 Presence of the Local APIC

Beginning with the P6 family processors, the presence or absence of an on-chip local APIC can be detected using the CPUID instruction. When the CPUID instruction is executed with a source operand of 1 in the EAX register, bit 9 of the CPUID feature flags returned in the EDX register indicates the presence (set) or absence (clear) of a local APIC.

10.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

- Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 10-5):
 - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 10.4.2, "Presence of the Local APIC") is also set to 0.
 - When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).
 - For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.
 - When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 10.4.7.1, "Local APIC State After Power-Up or Reset."
- Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23):
 - If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23). The state of the local APIC when in this software-disabled state is described in Section 10.4.7.2, "Local APIC State After It Has Been Software Disabled."
 - When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

10.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 10-5). MSR bit functions are described below:

- BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 8.4, "Multiple-Processor (MP) Initialization." Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 10.4.3, "Enabling or Disabling the Local APIC"). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.
- APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or reset, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR² through 63 in the IA32_APIC_BASE MSR are reserved.

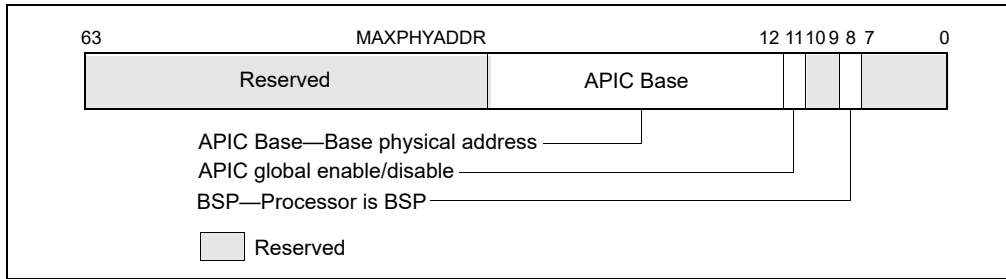


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

10.4.5 Relocating the Local APIC Registers

The Pentium 4, Intel Xeon, and P6 family processors permit the starting address of the APIC registers to be relocated from FEE00000H to another physical address by modifying the value in the base address field of the IA32_APIC_BASE MSR. This extension of the APIC architecture is provided to help resolve conflicts with memory maps of existing systems and to allow individual processors in an MP system to map their APIC registers to different locations in physical memory.

10.4.6 Local APIC ID

At power up, system hardware assigns a unique APIC ID to each local APIC on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for P6 family and Pentium processors). The hardware assigned APIC ID is based on system topology and includes encoding for socket position and cluster information (see Figure 8-2 and Section 8.9.1, "Hierarchical Mapping of Shared Resources").

In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. Some processors permit software to modify the APIC ID. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register. The value returned by bits 31-24 of the EBX register (when the CPUID instruction is executed with a source operand value of 1 in the EAX register) is always the Initial APIC ID (determined by the platform initialization). This is true even if software has changed the value in the Local APIC ID register.

The processor receives the hardware assigned APIC ID (or Initial APIC ID) by sampling pins A11# and A12# and pins BR0# through BR3# (for the Pentium 4, Intel Xeon, and P6 family processors) and pins BE0# through BE3# (for the Pentium processor). The APIC ID latched from these pins is stored in the APIC ID field of the local APIC ID register (see Figure 10-6), and is used as the Initial APIC ID for the processor.

2. The MAXPHYADDR is 36 bits for processors that do not support CPUID leaf 80000008H, or indicated by CPUID.80000008H:EAX[bits 7:0] for processors that support CPUID leaf 80000008H.

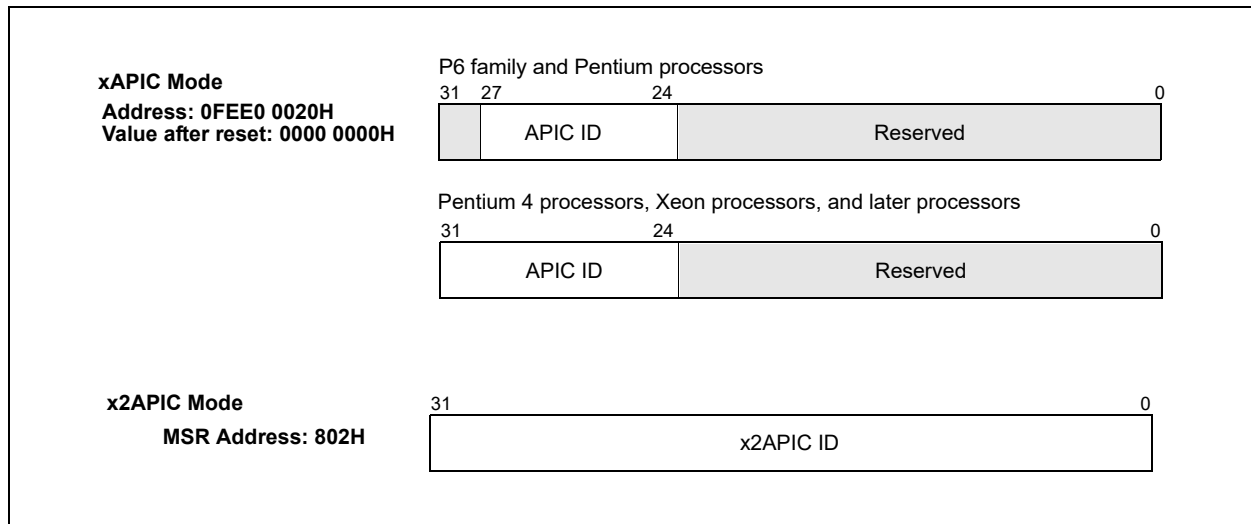


Figure 10-6. Local APIC ID Register

For the P6 family and Pentium processors, the local APIC ID field in the local APIC ID register is 4 bits. Encodings 0H through EH can be used to uniquely identify 15 different processors connected to the APIC bus. For the Pentium 4 and Intel Xeon processors, the xAPIC specification extends the local APIC ID field to 8 bits. These can be used to identify up to 255 processors in the system.

10.4.7 Local APIC State

The following sections describe the state of the local APIC and its registers following a power-up or reset, after the local APIC has been software disabled, following an INIT reset, and following an INIT-deassert message.

x2APIC will introduce 32-bit ID; see Section 10.12.

10.4.7.1 Local APIC State After Power-Up or Reset

Following a power-up or reset of the processor, the state of local APIC and its registers are as follows:

- The following registers are reset to all 0s.
 - IRR, ISR, TMR, ICR, LDR, and TPR.
 - Timer initial count and timer current count registers.
 - Divide configuration register.
- The DFR register is reset to all 1s.
- The LVT register is reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The local APIC ID register is set to a unique APIC ID. (Pentium and P6 family processors only). The Arb ID register is set to the value in the APIC ID register.
- The spurious-interrupt vector register is initialized to 000000FFH. By setting bit 8 to 0, software disables the local APIC.
- If the processor is the only processor in the system or it is the BSP in an MP system (see Section 8.4.1, “BSP and AP Processors”); the local APIC will respond normally to INIT and NMI messages, to INIT# signals and to STPCLK# signals. If the processor is in an MP system and has been designated as an AP; the local APIC will respond the same as for the BSP. In addition, it will respond to SIPI messages. For P6 family processors only, an AP will not respond to a STPCLK# signal.

10.4.7.2 Local APIC State After It Has Been Software Disabled

When the APIC software enable/disable flag in the spurious interrupt vector register has been explicitly cleared (as opposed to being cleared during a power up or reset), the local APIC is temporarily disabled (see Section 10.4.3, “Enabling or Disabling the Local APIC”). The operation and response of a local APIC while in this software-disabled state is as follows:

- The local APIC will respond normally to INIT, NMI, SMI, and SIPI messages.
- Pending interrupts in the IRR and ISR registers are held and require masking or handling by the CPU.
- The local APIC can still issue IPIs. It is software’s responsibility to avoid issuing IPIs through the IPI mechanism and the ICR register if sending interrupts through this mechanism is not desired.
- The reception of any interrupt or transmission of any IPIs that are in progress when the local APIC is disabled are completed before the local APIC enters the software-disabled state.
- The mask bits for all the LVT entries are set. Attempts to reset these bits will be ignored.
- (For Pentium and P6 family processors) The local APIC continues to listen to all bus messages in order to keep its arbitration ID synchronized with the rest of the system.

10.4.7.3 Local APIC State After an INIT Reset (“Wait-for-SIPI” State)

An INIT reset of the processor can be initiated in either of two ways:

- By asserting the processor’s INIT# pin.
- By sending the processor an INIT IPI (an IPI with the delivery mode set to INIT).

Upon receiving an INIT through either of these mechanisms, the processor responds by beginning the initialization process of the processor core and the local APIC. The state of the local APIC following an INIT reset is the same as it is after a power-up or hardware reset, except that the APIC ID and arbitration ID registers are not affected. This state is also referred to at the “wait-for-SIPI” state (see also: Section 8.4.2, “MP Initialization Protocol Requirements and Restrictions”).

10.4.7.4 Local APIC State After It Receives an INIT-Deassert IPI

Only the Pentium and P6 family processors support the INIT-deassert IPI. An INIT-deassert IPI has no effect on the state of the APIC, other than to reload the arbitration ID register with the value in the APIC ID register.

10.4.8 Local APIC Version Register

The local APIC contains a hardwired version register. Software can use this register to identify the APIC version (see Figure 10-7). In addition, the register specifies the number of entries in the local vector table (LVT) for a specific implementation.

The fields in the local APIC version register are as follows:

Version	The version numbers of the local APIC:
	0XH 82489DX discrete APIC.
	10H - 15H Integrated APIC.
	Other values reserved.
Max LVT Entry	Shows the number of LVT entries minus 1. For the Pentium 4 and Intel Xeon processors (which have 6 LVT entries), the value returned in the Max LVT field is 5; for the P6 family processors (which have 5 LVT entries), the value returned is 4; for the Pentium processor (which has 4 LVT entries), the value returned is 3. For processors based on the Intel microarchitecture code name Nehalem (which has 7 LVT entries) and onward, the value returned is 6.
Suppress EOI-broadcasts	Indicates whether software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register; see Section 10.8.5 and Section 10.9.

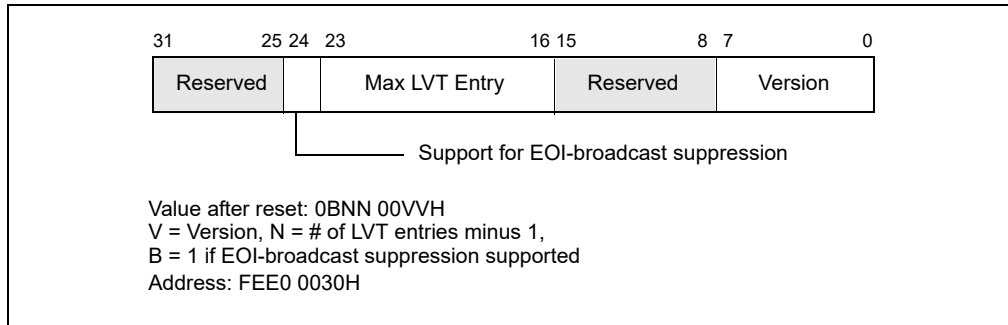


Figure 10-7. Local APIC Version Register

10.5 HANDLING LOCAL INTERRUPTS

The following sections describe facilities that are provided in the local APIC for handling local interrupts. These include: the processor’s LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector. Local interrupt handling facilities include: the LVT, the error status register (ESR), the divide configuration register (DCR), and the initial count and current count registers.

10.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 10-8), one for each local interrupt:

- **LVT CMCI Register (FEE0 02F0H)** — Specifies interrupt delivery when an overflow condition of corrected machine check error count reaching a threshold value occurred in a machine check bank supporting CMCI (see Section 15.5.1, “CMCI Local APIC Interface”).
- **LVT Timer Register (FEE0 0320H)** — Specifies interrupt delivery when the APIC timer signals an interrupt (see Section 10.5.4, “APIC Timer”).
- **LVT Thermal Monitor Register (FEE0 0330H)** — Specifies interrupt delivery when the thermal sensor generates an interrupt (see Section 14.8.2, “Thermal Monitor”). This LVT entry is implementation specific, not architectural. If implemented, it will always be at base address FEE0 0330H.
- **LVT Performance Counter Register (FEE0 0340H)** — Specifies interrupt delivery when a performance counter generates an interrupt on overflow (see Section 18.6.3.5.8, “Generating an Interrupt on Overflow”). This LVT entry is implementation specific, not architectural. If implemented, it is not guaranteed to be at base address FEE0 0340H.
- **LVT LINT0 Register (FEE0 0350H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT0 pin.
- **LVT LINT1 Register (FEE0 0360H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT1 pin.
- **LVT Error Register (FEE0 0370H)** — Specifies interrupt delivery when the APIC detects an internal error (see Section 10.5.3, “Error Handling”).

The LVT performance counter register and its associated interrupt were introduced in the P6 processors and are also present in the Pentium 4 and Intel Xeon processors. The LVT thermal monitor register and its associated interrupt were introduced in the Pentium 4 and Intel Xeon processors. The LVT CMCI register and its associated interrupt were introduced in the Intel Xeon 5500 processors.

As shown in Figures 10-8, some of these fields and flags are not available (and reserved) for some entries.

The setup information that can be specified in the registers of the LVT table is as follows:

Vector Interrupt vector number.

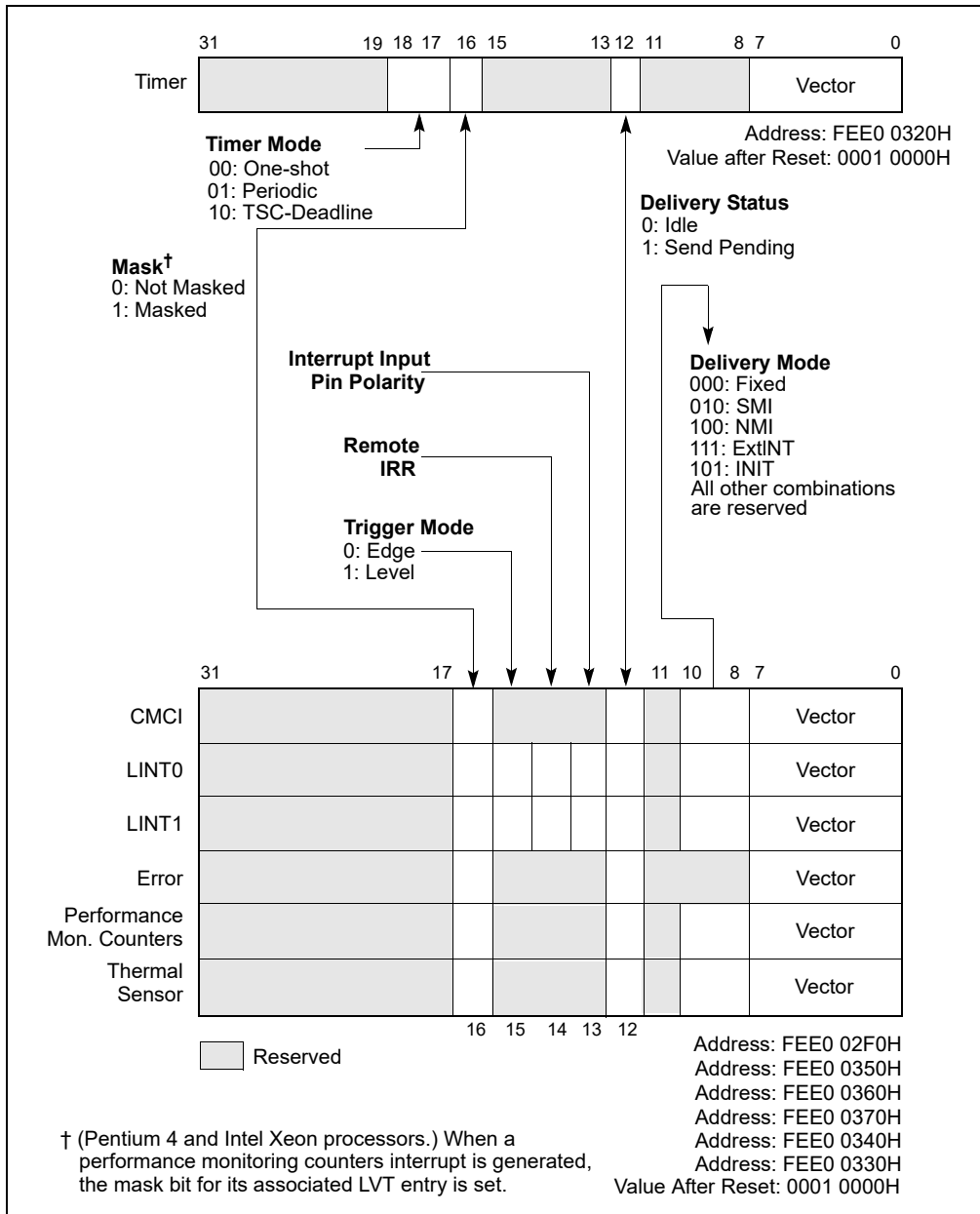


Figure 10-8. Local Vector Table (LVT)

Delivery Mode

Specifies the type of interrupt to be sent to the processor. Some delivery modes will only operate as intended when used in conjunction with a specific trigger mode. The allowable delivery modes are as follows:

- 000 (Fixed)** Delivers the interrupt specified in the vector field.
- 010 (SMI)** Delivers an SMI interrupt to the processor core through the processor’s local SMI signal path. When using this delivery mode, the vector field should be set to 00H for future compatibility.
- 100 (NMI)** Delivers an NMI interrupt to the processor. The vector information is ignored.
- 101 (INIT)** Delivers an INIT request to the processor core, which causes the processor to perform an INIT. When using this delivery mode, the vector field should

be set to 00H for future compatibility. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

110 Reserved; not supported for any LVT register.

111 (ExtINT) Causes the processor to respond to the interrupt as if the interrupt originated in an externally connected (8259A-compatible) interrupt controller. A special INTA bus cycle corresponding to ExtINT, is routed to the external controller. The external controller is expected to supply the vector information. The APIC architecture supports only one ExtINT source in a system, usually contained in the compatibility bridge. Only one processor in the system should have an LVT entry configured to use the ExtINT delivery mode. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

Delivery Status (Read Only)

Indicates the interrupt delivery status, as follows:

0 (Idle) There is currently no activity for this interrupt source, or the previous interrupt from this source was delivered to the processor core and accepted.

1 (Send Pending) Indicates that an interrupt from this source has been delivered to the processor core but has not yet been accepted (see Section 10.5.5, "Local Interrupt Acceptance").

Interrupt Input Pin Polarity

Specifies the polarity of the corresponding interrupt pin: (0) active high or (1) active low.

Remote IRR Flag (Read Only)

For fixed mode, level-triggered interrupts; this flag is set when the local APIC accepts the interrupt for servicing and is reset when an EOI command is received from the processor. The meaning of this flag is undefined for edge-triggered interrupts and other delivery modes.

Trigger Mode

Selects the trigger mode for the local LINT0 and LINT1 pins: (0) edge sensitive and (1) level sensitive. This flag is only used when the delivery mode is Fixed. When the delivery mode is NMI, SMI, or INIT, the trigger mode is always edge sensitive. When the delivery mode is ExtINT, the trigger mode is always level sensitive. The timer and error interrupts are always treated as edge sensitive.

If the local APIC is not used in conjunction with an I/O APIC and fixed delivery mode is selected; the Pentium 4, Intel Xeon, and P6 family processors will always use level-sensitive triggering, regardless if edge-sensitive triggering is selected.

Software should always set the trigger mode in the LVT LINT1 register to 0 (edge sensitive). Level-sensitive interrupts are not supported for LINT1.

Mask

Interrupt mask: (0) enables reception of the interrupt and (1) inhibits reception of the interrupt. When the local APIC handles a performance-monitoring counters interrupt, it automatically sets the mask flag in the LVT performance counter register. This flag is set to 1 on reset. It can be cleared only by software.

Timer Mode

Bits 18:17 selects the timer mode (see Section 10.5.4):

(00b) one-shot mode using a count-down value,

(01b) periodic mode reloading a count-down value,

(10b) TSC-Deadline mode using absolute target value in IA32_TSC_DEADLINE MSR (see Section 10.5.4.1),

(11b) is reserved.

10.5.2 Valid Interrupt Vectors

The Intel 64 and IA-32 architectures define 256 vector numbers, ranging from 0 through 255 (see Section 6.2, "Exception and Interrupt Vectors"). Local and I/O APICs support 240 of these vectors (in the range of 16 to 255) as valid interrupts.

When an interrupt vector in the range of 0 to 15 is sent or received through the local APIC, the APIC indicates an illegal vector in its Error Status Register (see Section 10.5.3, "Error Handling"). The Intel 64 and IA-32 architectures reserve vectors 16 through 31 for predefined interrupts, exceptions, and Intel-reserved encodings (see Table 6-1). However, the local APIC does not treat vectors in this range as illegal.

When an illegal vector value (0 to 15) is written to an LVT entry and the delivery mode is Fixed (bits 8-11 equal 0), the APIC may signal an illegal vector error, without regard to whether the mask bit is set or whether an interrupt is actually seen on the input.

10.5.3 Error Handling

The local APIC records errors detected during interrupt handling in the error status register (ESR). The format of the ESR is given in Figure 10-9; it contains the following flags:

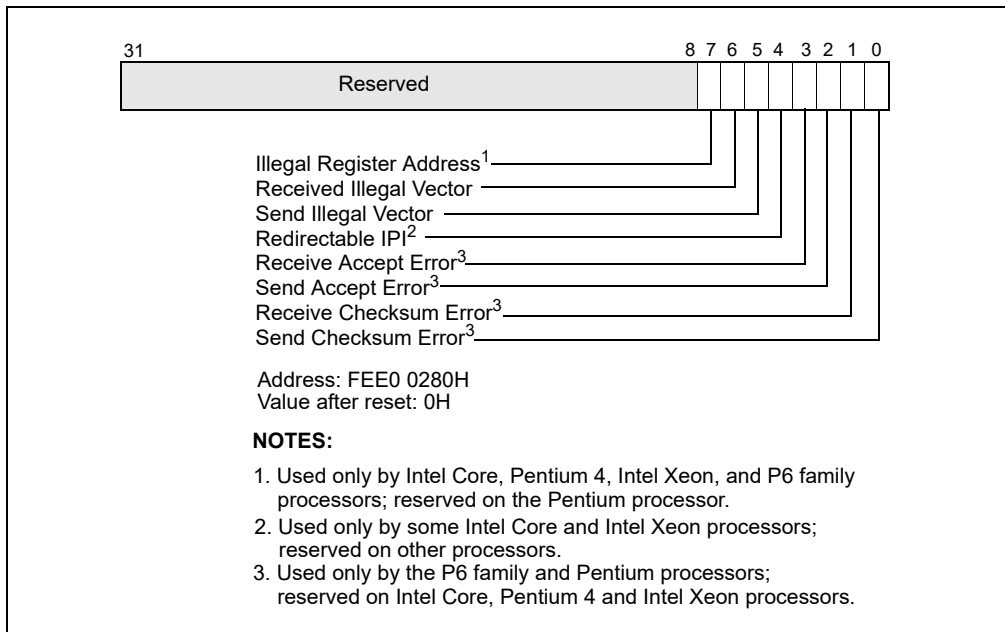


Figure 10-9. Error Status Register (ESR)

- **Bit 0: Send Checksum Error.**
Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. Used only on P6 family and Pentium processors.
- **Bit 1: Receive Checksum Error.**
Set when the local APIC detects a checksum error for a message that it received on the APIC bus. Used only on P6 family and Pentium processors.
- **Bit 2: Send Accept Error.**
Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus. Used only on P6 family and Pentium processors.
- **Bit 3: Receive Accept Error.**
Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself. Used only on P6 family and Pentium processors.
- **Bit 4: Redirectable IPI.**
Set when the local APIC detects an attempt to send an IPI with the lowest-priority delivery mode and the local APIC does not support the sending of such IPIs. This bit is used on some Intel Core and Intel Xeon processors. As noted in Section 10.6.2, the ability of a processor to send a lowest-priority IPI is model-specific and should be avoided.

- Bit 5: Send Illegal Vector.**
Set when the local APIC detects an illegal vector (one in the range 0 to 15) in the message that it is sending. This occurs as the result of a write to the ICR (in both xAPIC and x2APIC modes) or to SELF IPI register (x2APIC mode only) with an illegal vector.

If the local APIC does not support the sending of lowest-priority IPIs and software writes the ICR to send a lowest-priority IPI with an illegal vector, the local APIC sets only the “redirectable IPI” error bit. The interrupt is not processed and hence the “Send Illegal Vector” bit is not set in the ESR.
- Bit 6: Receive Illegal Vector.**
Set when the local APIC detects an illegal vector (one in the range 0 to 15) in an interrupt message it receives or in an interrupt generated locally from the local vector table or via a self IPI. Such interrupts are not delivered to the processor; the local APIC will never set an IRR bit in the range 0 to 15.
- Bit 7: Illegal Register Address**
Set when the local APIC is in xAPIC mode and software attempts to access a register that is reserved in the processor's local-APIC register-address space; see Table 10-1. (The local-APIC register-address space comprises the 4 KBytes at the physical address specified in the IA32_APIC_BASE MSR.) Used only on Intel Core, Intel Atom™, Pentium 4, Intel Xeon, and P6 family processors.

In x2APIC mode, software accesses the APIC registers using the RDMSR and WRMSR instructions. Use of one of these instructions to access a reserved register cause a general-protection exception (see Section 10.12.1.3). They do not set the “Illegal Register Access” bit in the ESR.

The ESR is a write/read register. Before attempt to read from the ESR, software should first write to it. (The value written does not affect the values read subsequently; only zero may be written in x2APIC mode.) This write clears any previously logged errors and updates the ESR with any errors detected since the last write to the ESR. This write also rearms the APIC error interrupt triggering mechanism.

The LVT Error Register (see Section 10.5.1) allows specification of the vector of the interrupt to be delivered to the processor core when APIC error is detected. The register also provides a means of masking an APIC-error interrupt. This masking only prevents delivery of APIC-error interrupts; the APIC continues to record errors in the ESR.

10.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 10-10), the initial-count and current-count registers (see Figure 10-11), and the LVT timer register (see Figure 10-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor’s APIC timer runs at a constant rate regardless of P-state transitions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

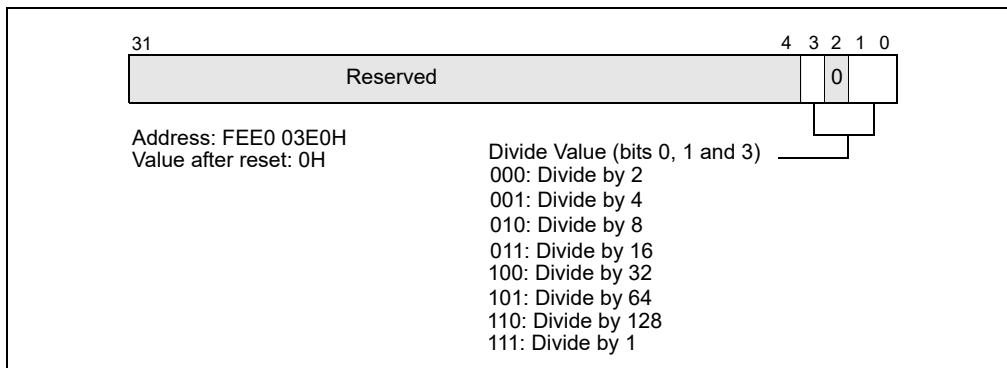


Figure 10-10. Divide Configuration Register

The APIC timer frequency will be the processor’s bus clock or core crystal clock frequency (when TSC/core crystal clock ratio is enumerated in CPUID leaf 0x15) divided by the value specified in the divide configuration register.

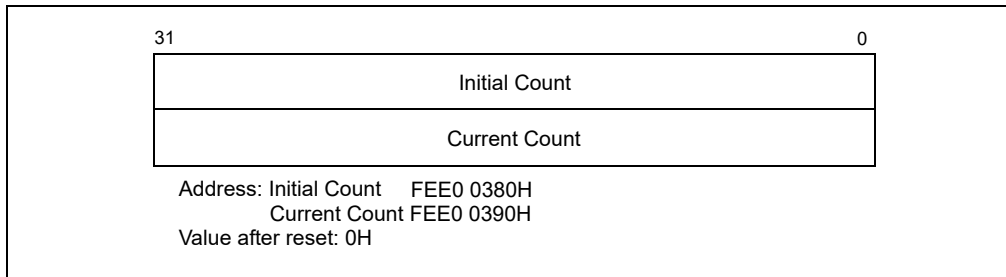


Figure 10-11. Initial Count and Current Count Registers

The timer can be configured through the timer LVT entry for one-shot or periodic operation. In one-shot mode, the timer is started by programming its initial-count register. The initial count value is then copied into the current-count register and count-down begins. After the timer reaches zero, a timer interrupt is generated and the timer remains at its 0 value until reprogrammed.

In periodic mode, the current-count register is automatically reloaded from the initial-count register when the count reaches 0 and a timer interrupt is generated, and the count-down is repeated. If during the count-down process the initial-count register is set, counting will restart, using the new initial-count value. The initial-count register is a read-write register; the current-count register is read only.

A write of 0 to the initial-count register effectively stops the local APIC timer, in both one-shot and periodic mode.

The LVT timer register determines the vector number that is delivered to the processor with the timer interrupt that is generated when the timer count reaches zero. The mask flag in the LVT timer register can be used to mask the timer interrupt.

10.5.4.1 TSC-Deadline Mode

The mode of operation of the local-APIC timer is determined by the LVT Timer Register. Specifically:

- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the mode is determined by bit 17 of the register.
- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the mode is determined by bits 18:17. See Figure 10-8. (If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, bit 18 of the register is reserved.)

The supported timer modes are given in Table 10-2. The three modes of the local APIC timer are mutually exclusive.

Table 10-2. Local APIC Timer Modes

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

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

The IA32_TSC_DEADLINE MSR (MSR address 6E0H) is a per-logical processor MSR that specifies the time at which a timer interrupt should occur. Writing a non-zero 64-bit value into IA32_TSC_DEADLINE arms the timer. An interrupt is generated when the logical processor's time-stamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR.³ When the timer generates an interrupt, it disarms itself and clears the IA32_TSC_DEADLINE MSR. Thus, each write to the IA32_TSC_DEADLINE MSR generates at most one timer interrupt.

In TSC-deadline mode, writing 0 to the IA32_TSC_DEADLINE MSR disarms the local-APIC timer. Transitioning between TSC-deadline mode and other timer modes also disarms the timer.

The hardware reset value of the IA32_TSC_DEADLINE MSR is 0. In other timer modes (LVT bit 18 = 0), the IA32_TSC_DEADLINE MSR reads zero and writes are ignored.

Software can configure the TSC-deadline timer to deliver a single interrupt using the following algorithm:

1. Detect support for TSC-deadline mode by verifying CPUID.1:ECX.24 = 1.
2. Select the TSC-deadline mode by programming bits 18:17 of the LVT Timer register with 10b.
3. Program the IA32_TSC_DEADLINE MSR with the target TSC value at which the timer interrupt is desired. This causes the processor to arm the timer.
4. The processor generates a timer interrupt when the value of time-stamp counter is greater than or equal to that of IA32_TSC_DEADLINE. It then disarms the timer and clear the IA32_TSC_DEADLINE MSR. (Both the time-stamp counter and the IA32_TSC_DEADLINE MSR are 64-bit unsigned integers.)
5. Software can re-arm the timer by repeating step 3.

The following are usage guidelines for TSC-deadline mode:

- Writes to the IA32_TSC_DEADLINE MSR are not serialized. Therefore, system software should not use WRMSR to the IA32_TSC_DEADLINE MSR as a serializing instruction. Read and write accesses to the IA32_TSC_DEADLINE and other MSR registers will occur in program order.
- Software can disarm the timer at any time by writing 0 to the IA32_TSC_DEADLINE MSR.
- If timer is armed, software can change the deadline (forward or backward) by writing a new value to the IA32_TSC_DEADLINE MSR.
- If software disarms the timer or postpones the deadline, race conditions may result in the delivery of a spurious timer interrupt. Software is expected to detect such spurious interrupts by checking the current value of the time-stamp counter to confirm that the interrupt was desired.⁴
- In xAPIC mode (in which the local-APIC registers are memory-mapped), software must order the memory-mapped write to the LVT entry that enables TSC-deadline mode and any subsequent WRMSR to the IA32_TSC_DEADLINE MSR. Software can assure proper ordering by executing the MFENCE instruction after the memory-mapped write and before any WRMSR. (In x2APIC mode, the WRMSR instruction is used to write to the LVT entry. The processor ensures the ordering of this write and any subsequent WRMSR to the deadline; no fencing is required.)

10.5.5 Local Interrupt Acceptance

When a local interrupt is sent to the processor core, it is subject to the acceptance criteria specified in the interrupt acceptance flow chart in Figure 10-17. If the interrupt is accepted, it is logged into the IRR register and handled by the processor according to its priority (see Section 10.8.4, "Interrupt Acceptance for Fixed Interrupts"). If the interrupt is not accepted, it is sent back to the local APIC and retried.

10.6 ISSUING INTERPROCESSOR INTERRUPTS

The following sections describe the local APIC facilities that are provided for issuing interprocessor interrupts (IPIs) from software. The primary local APIC facility for issuing IPIs is the interrupt command register (ICR). The ICR can be used for the following functions:

-
3. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.
 4. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.

- To send an interrupt to another processor.
- To allow a processor to forward an interrupt that it received but did not service to another processor for servicing.
- To direct the processor to interrupt itself (perform a self interrupt).
- To deliver special IPIs, such as the start-up IPI (SIPI) message, to other processors.

Interrupts generated with this facility are delivered to the other processors in the system through the system bus (for Pentium 4 and Intel Xeon processors) or the APIC bus (for P6 family and Pentium processors). The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

10.6.1 Interrupt Command Register (ICR)

The interrupt command register (ICR) is a 64-bit⁵ local APIC register (see Figure 10-12) that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system.

To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. (All fields of the ICR are read-write by software with the exception of the delivery status field, which is read-only.) The act of writing to the low doubleword of the ICR causes the IPI to be sent.

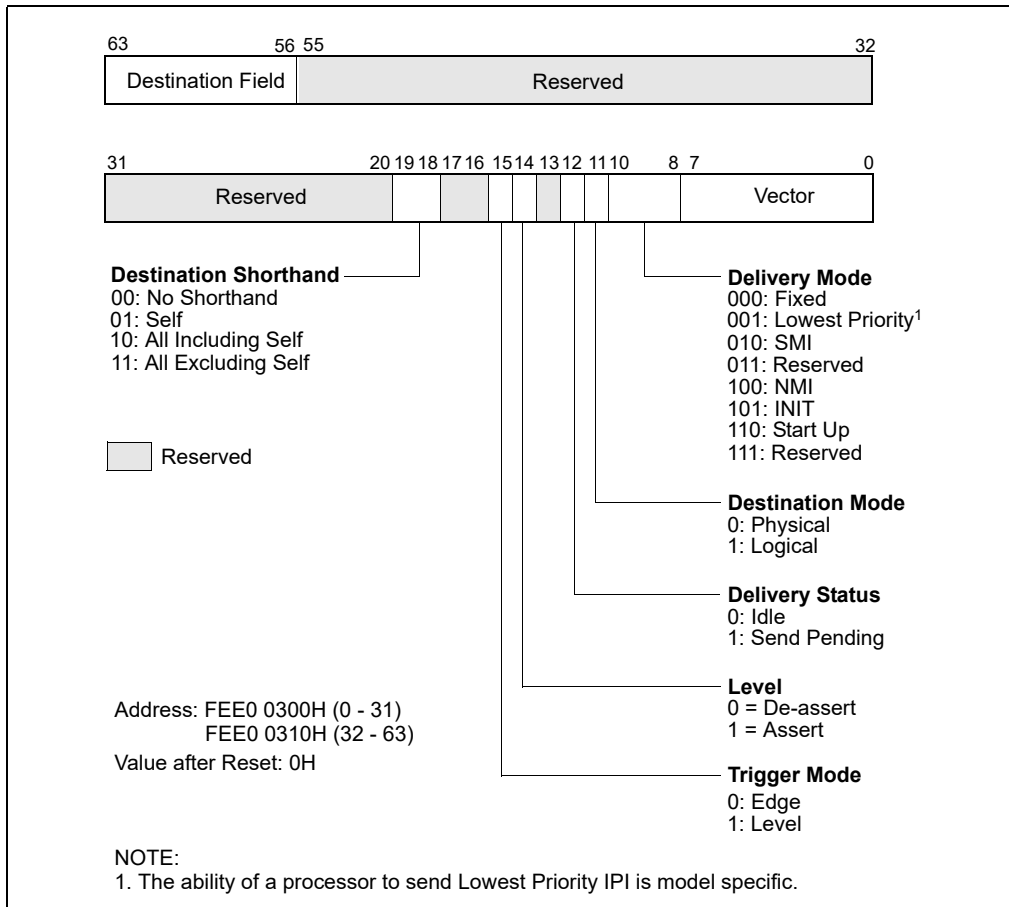


Figure 10-12. Interrupt Command Register (ICR)

5. In XAPIC mode the ICR is addressed as two 32-bit registers, ICR_LOW (FEE0 0300H) and ICR_HIGH (FEE0 0310H). In x2APIC mode, the ICR uses MSR 830H.

The ICR consists of the following fields.

Vector	The vector number of the interrupt being sent.
Delivery Mode	<p>Specifies the type of IPI to be sent. This field is also know as the IPI message type field.</p> <p>000 (Fixed) Delivers the interrupt specified in the vector field to the target processor or processors.</p> <p>001 (Lowest Priority) Same as fixed mode, except that the interrupt is delivered to the processor executing at the lowest priority among the set of processors specified in the destination field. The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.</p> <p>010 (SMI) Delivers an SMI interrupt to the target processor or processors. The vector field must be programmed to 00H for future compatibility.</p> <p>011 (Reserved)</p> <p>100 (NMI) Delivers an NMI interrupt to the target processor or processors. The vector information is ignored.</p> <p>101 (INIT) Delivers an INIT request to the target processor or processors, which causes them to perform an INIT. As a result of this IPI message, all the target processors perform an INIT. The vector field must be programmed to 00H for future compatibility.</p> <p>101 (INIT Level De-assert) (Not supported in the Pentium 4 and Intel Xeon processors.) Sends a synchronization message to all the local APICs in the system to set their arbitration IDs (stored in their Arb ID registers) to the values of their APIC IDs (see Section 10.7, "System and APIC Bus Arbitration"). For this delivery mode, the level flag must be set to 0 and trigger mode flag to 1. This IPI is sent to all processors, regardless of the value in the destination field or the destination shorthand field; however, software should specify the "all including self" shorthand.</p> <p>110 (Start-Up) Sends a special "start-up" IPI (called a SIPI) to the target processor or processors. The vector typically points to a start-up routine that is part of the BIOS boot-strap code (see Section 8.4, "Multiple-Processor (MP) Initialization"). IPIs sent with this delivery mode are not automatically retried if the source APIC is unable to deliver it. It is up to the software to determine if the SIPI was not successfully delivered and to reissue the SIPI if necessary.</p>
Destination Mode	Selects either physical (0) or logical (1) destination mode (see Section 10.6.2, "Determining IPI Destination").
Delivery Status (Read Only)	<p>Indicates the IPI delivery status, as follows:</p> <p>0 (Idle) Indicates that this local APIC has completed sending any previous IPIs.</p> <p>1 (Send Pending) Indicates that this local APIC has not completed sending the last IPI.</p>
Level	For the INIT level de-assert delivery mode this flag must be set to 0; for all other delivery modes it must be set to 1. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 1.)

Trigger Mode Selects the trigger mode when using the INIT level de-assert delivery mode: edge (0) or level (1). It is ignored for all other delivery modes. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 0.)

Destination Shorthand

Indicates whether a shorthand notation is used to specify the destination of the interrupt and, if so, which shorthand is used. Destination shorthands are used in place of the 8-bit destination field, and can be sent by software using a single write to the low doubleword of the ICR. Shorthands are defined for the following cases: software self interrupt, IPIs to all processors in the system including the sender, IPIs to all processors in the system excluding the sender.

00: (No Shorthand)

The destination is specified in the destination field.

01: (Self)

The issuing APIC is the one and only destination of the IPI. This destination shorthand allows software to interrupt the processor on which it is executing. An APIC implementation is free to deliver the self-interrupt message internally or to issue the message to the bus and “snoop” it as with any other IPI message.

10: (All Including Self)

The IPI is sent to all processors in the system including the processor sending the IPI. The APIC will broadcast an IPI message with the destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors.

11: (All Excluding Self)

The IPI is sent to all processors in a system with the exception of the processor sending the IPI. The APIC broadcasts a message with the physical destination mode and destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors. Support for this destination shorthand in conjunction with the lowest-priority delivery mode is model specific. For Pentium 4 and Intel Xeon processors, when this shorthand is used together with lowest priority delivery mode, the IPI may be redirected back to the issuing processor.

Destination

Specifies the target processor or processors. This field is only used when the destination shorthand field is set to 00B. If the destination mode is set to physical, then bits 56 through 59 contain the APIC ID of the target processor for Pentium and P6 family processors and bits 56 through 63 contain the APIC ID of the target processor the for Pentium 4 and Intel Xeon processors. If the destination mode is set to logical, the interpretation of the 8-bit destination field depends on the settings of the DFR and LDR registers of the local APICs in all the processors in the system (see Section 10.6.2, “Determining IPI Destination”).

Not all combinations of options for the ICR are valid. Table 10-3 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 10-4 shows the valid combinations for the fields in the ICR for the P6 family processors. Also note that the lower half of the ICR may not be preserved over transitions to the deepest C-States.

ICR operation in x2APIC mode is discussed in Section 10.12.9.

Table 10-3 Valid Combinations for the Pentium 4 and Intel Xeon Processors’ Local xAPIC Interrupt Command Register

Destination Shorthand	Valid/Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes ¹	Physical or Logical
No Shorthand	Invalid ²	Level	All Modes	Physical or Logical
Self	Valid	Edge	Fixed	X ³
Self	Invalid ²	Level	Fixed	X
Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Including Self	Valid	Edge	Fixed	X
All Including Self	Invalid ²	Level	Fixed	X
All Including Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Excluding Self	Valid	Edge	Fixed, Lowest Priority ^{1,4} , NMI, INIT, SMI, Start-Up	X
All Excluding Self	Invalid ²	Level	Fixed, Lowest Priority ⁴ , NMI, INIT, SMI, Start-Up	X

NOTES:

1. The ability of a processor to send a lowest priority IPI is model specific.
2. For these interrupts, if the trigger mode bit is 1 (Level), the local xAPIC will override the bit setting and issue the interrupt as an edge triggered interrupt.
3. X means the setting is ignored.
4. When using the “lowest priority” delivery mode and the “all excluding self” destination, the IPI can be redirected back to the issuing APIC, which is essentially the same as the “all including self” destination mode.

Table 10-4 Valid Combinations for the P6 Family Processors’ Local APIC Interrupt Command Register

Destination Shorthand	Valid/Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes ¹	Physical or Logical
No Shorthand	Valid ²	Level	Fixed, Lowest Priority ¹ , NMI	Physical or Logical
No Shorthand	Valid ³	Level	INIT	Physical or Logical
Self	Valid	Edge	Fixed	X ⁴
Self	Valid ²	Level	Fixed	X
Self	Invalid ⁵	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All including Self	Valid	Edge	Fixed	X
All including Self	Valid ²	Level	Fixed	X
All including Self	Invalid ⁵	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All excluding Self	Valid	Edge	All Modes ¹	X
All excluding Self	Valid ²	Level	Fixed, Lowest Priority ¹ , NMI	X
All excluding Self	Invalid ⁵	Level	SMI, Start-Up	X
All excluding Self	Valid ³	Level	INIT	X
X	Invalid ⁵	Level	SMI, Start-Up	X

NOTES:

1. The ability of a processor to send a lowest priority IPI is model specific.
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.
3. Treated as edge triggered when Level bit is set to 1; treated as “INIT Level Deassert” message when level bit is set to 0 (deassert). Only INIT level deassert messages are allowed to have the level bit set to 0. For all other messages the level bit must be set to 1.
4. X means the setting is ignored.
5. The behavior of the APIC is undefined.

10.6.2 Determining IPI Destination

The destination of an IPI⁶ can be one, all, or a subset (group) of the processors on the system bus. The sender of the IPI specifies the destination of an IPI with the following APIC registers and fields within the registers:

- **ICR Register** — The following fields in the ICR register are used to specify the destination of an IPI.
 - **Destination Mode** — Selects one of two destination modes (physical or logical).
 - **Destination Field** — In physical destination mode, used to specify the APIC ID of the destination processor; in logical destination mode, used to specify a message destination address (MDA) that can be used to select specific processors in clusters.
 - **Destination Shorthand** — A quick method of specifying all processors, all excluding self, or self as the destination.
 - **Delivery mode, Lowest Priority** — Architecturally specifies that a lowest-priority arbitration mechanism be used to select a destination processor from a specified group of processors. The ability of a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.
- **Local destination register (LDR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.
- **Destination format register (DFR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

How the ICR, LDR, and DFR are used to select an IPI destination depends on the destination mode used: physical, logical, broadcast/self, or lowest-priority delivery mode. These destination modes are described in the following sections.

10.6.2.1 Physical Destination Mode

In physical destination mode, the destination processor is specified by its local APIC ID (see Section 10.4.6, “Local APIC ID”). For Pentium 4 and Intel Xeon processors, either a single destination (local APIC IDs 00H through FEH) or a broadcast to all APICs (the APIC ID is FFH) may be specified in physical destination mode.

A broadcast IPI (bits 28-31 of the MDA are 1's) or I/O subsystem initiated interrupt with lowest priority delivery mode is not supported in physical destination mode and must not be configured by software. Also, for any non-broadcast IPI or I/O subsystem initiated interrupt with lowest priority delivery mode, software must ensure that APICs defined in the interrupt address are present and enabled to receive interrupts.

For the P6 family and Pentium processors, a single destination is specified in physical destination mode with a local APIC ID of 0H through 0EH, allowing up to 15 local APICs to be addressed on the APIC bus. A broadcast to all local APICs is specified with 0FH.

NOTE

The number of local APICs that can be addressed on the system bus may be restricted by hardware.

10.6.2.2 Logical Destination Mode

In logical destination mode, IPI destination is specified using an 8-bit message destination address (MDA), which is entered in the destination field of the ICR. Upon receiving an IPI message that was sent using logical destination mode, a local APIC compares the MDA in the message with the values in its LDR and DFR to determine if it should accept and handle the IPI. For both configurations of logical destination mode, when combined with lowest priority delivery mode, software is responsible for ensuring that all of the local APICs included in or addressed by the IPI or I/O subsystem interrupt are present and enabled to receive the interrupt.

Figure 10-13 shows the layout of the logical destination register (LDR). The 8-bit logical APIC ID field in this register is used to create an identifier that can be compared with the MDA.

6. Determination of IPI destinations in x2APIC mode is discussed in Section 10.12.10.

NOTE

The logical APIC ID should not be confused with the local APIC ID that is contained in the local APIC ID register.

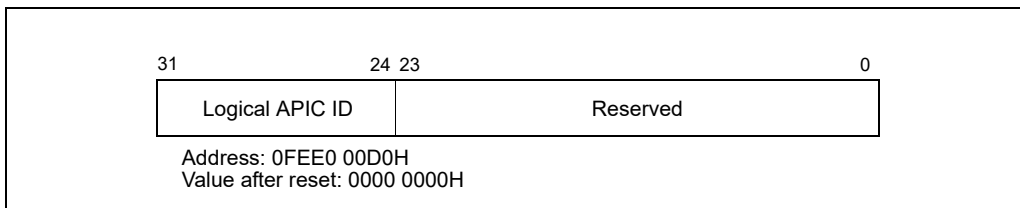


Figure 10-13. Logical Destination Register (LDR)

Figure 10-14 shows the layout of the destination format register (DFR). The 4-bit model field in this register selects one of two models (flat or cluster) that can be used to interpret the MDA when using logical destination mode.

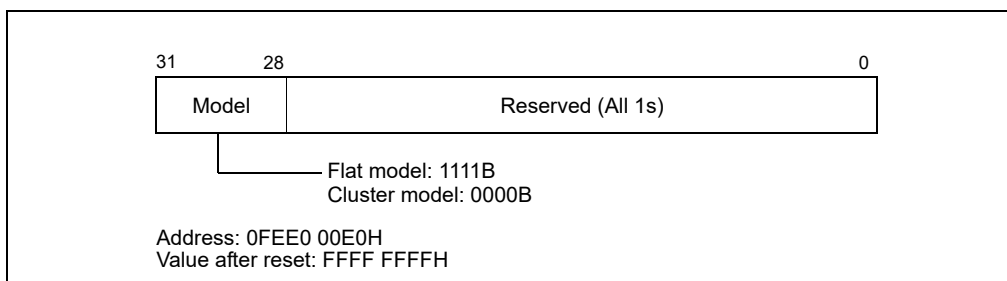


Figure 10-14. Destination Format Register (DFR)

The interpretation of MDA for the two models is described in the following paragraphs.

1. **Flat Model** — This model is selected by programming DFR bits 28 through 31 to 1111. Here, a unique logical APIC ID can be established for up to 8 local APICs by setting a different bit in the logical APIC ID field of the LDR for each local APIC. A group of local APICs can then be selected by setting one or more bits in the MDA.

Each local APIC performs a bit-wise AND of the MDA and its logical APIC ID. If a true condition (non-zero) is detected, the local APIC accepts the IPI message. A broadcast to all APICs is achieved by setting the MDA to 1s.

2. **Cluster Model** — This model is selected by programming DFR bits 28 through 31 to 0000. This model supports two basic destination schemes: flat cluster and hierarchical cluster.

The flat cluster destination model is only supported for P6 family and Pentium processors. Using this model, all APICs are assumed to be connected through the APIC bus. Bits 60 through 63 of the MDA contains the encoded address of the destination cluster and bits 56 through 59 identify up to four local APICs within the cluster (each bit is assigned to one local APIC in the cluster, as in the flat connection model). To identify one or more local APICs, bits 60 through 63 of the MDA are compared with bits 28 through 31 of the LDR to determine if a local APIC is part of the cluster. Bits 56 through 59 of the MDA are compared with Bits 24 through 27 of the LDR to identify a local APICs within the cluster.

Sets of processors within a cluster can be specified by writing the target cluster address in bits 60 through 63 of the MDA and setting selected bits in bits 56 through 59 of the MDA, corresponding to the chosen members of the cluster. In this mode, 15 clusters (with cluster addresses of 0 through 14) each having 4 local APICs can be specified in the message. For the P6 and Pentium processor’s local APICs, however, the APIC arbitration ID supports only 15 APIC agents. Therefore, the total number of processors and their local APICs supported in this mode is limited to 15. Broadcast to all local APICs is achieved by setting all destination bits to one. This guarantees a match on all clusters and selects all APICs in each cluster. A broadcast IPI or I/O subsystem broadcast interrupt with lowest priority delivery mode is not supported in cluster mode and must not be configured by software.

The hierarchical cluster destination model can be used with Pentium 4, Intel Xeon, P6 family, or Pentium processors. With this model, a hierarchical network can be created by connecting different flat clusters via

independent system or APIC buses. This scheme requires a cluster manager within each cluster, which is responsible for handling message passing between system or APIC buses. One cluster contains up to 4 agents. Thus 15 cluster managers, each with 4 agents, can form a network of up to 60 APIC agents. Note that hierarchical APIC networks requires a special cluster manager device, which is not part of the local or the I/O APIC units.

NOTES

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.
 The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

10.6.2.3 Broadcast/Self Delivery Mode

The destination shorthand field of the ICR allows the delivery mode to be by-passed in favor of broadcasting the IPI to all the processors on the system bus and/or back to itself (see Section 10.6.1, "Interrupt Command Register (ICR)"). Three destination shorthands are supported: self, all excluding self, and all including self. The destination mode is ignored when a destination shorthand is used.

10.6.2.4 Lowest Priority Delivery Mode

With lowest priority delivery mode, the ICR is programmed to send an IPI to several processors on the system bus, using the logical or shorthand destination mechanism for selecting the processor. The selected processors then arbitrate with one another over the system bus or the APIC bus, with the lowest-priority processor accepting the IPI.

For systems based on the Intel Xeon processor, the chipset bus controller accepts messages from the I/O APIC agents in the system and directs interrupts to the processors on the system bus. When using the lowest priority delivery mode, the chipset chooses a target processor to receive the interrupt out of the set of possible targets. The Pentium 4 processor provides a special bus cycle on the system bus that informs the chipset of the current task priority for each logical processor in the system. The chipset saves this information and uses it to choose the lowest priority processor when an interrupt is received.

For systems based on P6 family processors, the processor priority used in lowest-priority arbitration is contained in the arbitration priority register (APR) in each local APIC. Figure 10-15 shows the layout of the APR.

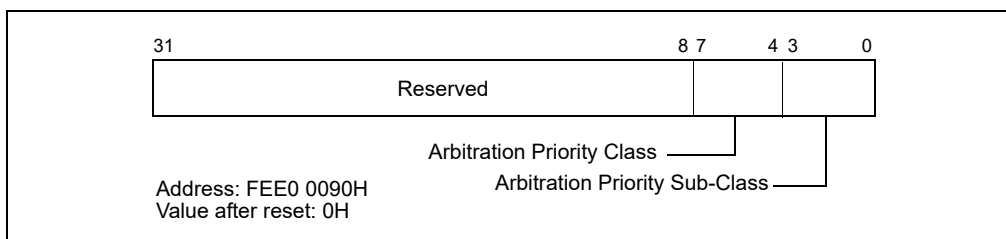


Figure 10-15. Arbitration Priority Register (APR)

The APR value is computed as follows:

```

IF (TPR[7:4] ≥ IRRV[7:4]) AND (TPR[7:4] > ISRV[7:4])
    THEN
        APR[7:0] ← TPR[7:0]
    ELSE
        APR[7:4] ← max(TPR[7:4] AND ISRV[7:4], IRRV[7:4])
        APR[3:0] ← 0.
    
```

Here, the TPR value is the task priority value in the TPR (see Figure 10-18), the IRRV value is the vector number for the highest priority bit that is set in the IRR (see Figure 10-20) or 00H (if no IRR bit is set), and the ISRV value is the vector number for the highest priority bit that is set in the ISR (see Figure 10-20). Following arbitration among the destination processors, the processor with the lowest value in its APR handles the IPI and the other processors ignore it.

(P6 family and Pentium processors.) For these processors, if a **focus processor** exists, it may accept the interrupt, regardless of its priority. A processor is said to be the focus of an interrupt if it is currently servicing that interrupt or if it has a pending request for that interrupt. For Intel Xeon processors, the concept of a focus processor is not supported.

In operating systems that use the lowest priority delivery mode but do not update the TPR, the TPR information saved in the chipset will potentially cause the interrupt to be always delivered to the same processor from the logical set. This behavior is functionally backward compatible with the P6 family processor but may result in unexpected performance implications.

10.6.3 IPI Delivery and Acceptance

When the low double-word of the ICR is written to, the local APIC creates an IPI message from the information contained in the ICR and sends the message out on the system bus (Pentium 4 and Intel Xeon processors) or the APIC bus (P6 family and Pentium processors). The manner in which these IPIs are handled after being issues in described in Section 10.8, "Handling Interrupts."

10.7 SYSTEM AND APIC BUS ARBITRATION

When several local APICs and the I/O APIC are sending IPI and interrupt messages on the system bus (or APIC bus), the order in which the messages are sent and handled is determined through bus arbitration.

For the Pentium 4 and Intel Xeon processors, the local and I/O APICs use the arbitration mechanism defined for the system bus to determine the order in which IPIs are handled. This mechanism is non-architectural and cannot be controlled by software.

For the P6 family and Pentium processors, the local and I/O APICs use an APIC-based arbitration mechanism to determine the order in which IPIs are handled. Here, each local APIC is given an arbitration priority of from 0 to 15, which the I/O APIC uses during arbitration to determine which local APIC should be given access to the APIC bus. The local APIC with the highest arbitration priority always wins bus access. Upon completion of an arbitration round, the winning local APIC lowers its arbitration priority to 0 and the losing local APICs each raise theirs by 1.

The current arbitration priority for a local APIC is stored in a 4-bit, software-transparent arbitration ID (Arb ID) register. During reset, this register is initialized to the APIC ID number (stored in the local APIC ID register). The INIT level-deassert IPI, which is issued with an ICR command, can be used to resynchronize the arbitration priorities of the local APICs by resetting Arb ID register of each agent to its current APIC ID value. (The Pentium 4 and Intel Xeon processors do not implement the Arb ID register.)

Section 10.10, "APIC Bus Message Passing Mechanism and Protocol (P6 Family, Pentium Processors)," describes the APIC bus arbitration protocols and bus message formats, while Section 10.6.1, "Interrupt Command Register (ICR)," describes the INIT level de-assert IPI message.

Note that except for the SIPI IPI (see Section 10.6.1, "Interrupt Command Register (ICR)"), all bus messages that fail to be delivered to their specified destination or destinations are automatically retried. Software should avoid situations in which IPIs are sent to disabled or nonexistent local APICs, causing the messages to be resent repeatedly. Additionally, interrupt sources that target the APIC should be masked or changed to no longer target the APIC.

10.8 HANDLING INTERRUPTS

When a local APIC receives an interrupt from a local source, an interrupt message from an I/O APIC, or an IPI, the manner in which it handles the message depends on processor implementation, as described in the following sections.

10.8.1 Interrupt Handling with the Pentium 4 and Intel Xeon Processors

With the Pentium 4 and Intel Xeon processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows:

1. It determines if it is the specified destination or not (see Figure 10-16). If it is the specified destination, it accepts the message; if it is not, it discards the message.

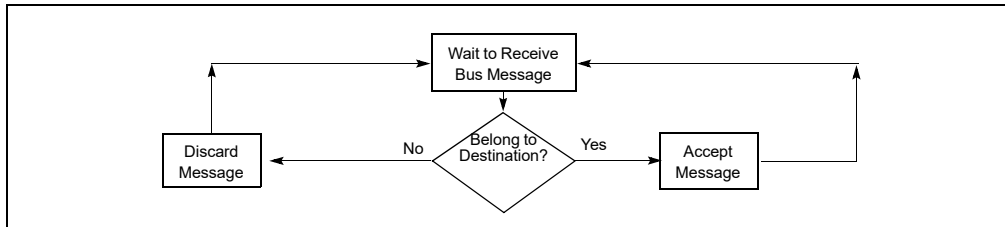


Figure 10-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors)

2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or SIPI, the interrupt is sent directly to the processor core for handling.
3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC sets the appropriate bit in the IRR.
4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 10.8.3.1, "Task and Processor Priorities").
5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC (see Section 10.8.5, "Signaling Interrupt Servicing Completion"). The act of writing to the EOI register causes the local APIC to delete the interrupt from its ISR queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

10.8.2 Interrupt Handling with the P6 Family and Pentium Processors

With the P6 family and Pentium processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows (see Figure 10-17).

1. (IPIs only) The local APIC examines the IPI message to determine if it is the specified destination for the IPI as described in Section 10.6.2, "Determining IPI Destination." If it is the specified destination, it continues its acceptance procedure; if it is not the destination, it discards the IPI message. When the message specifies lowest-priority delivery mode, the local APIC will arbitrate with the other processors that were designated as recipients of the IPI message (see Section 10.6.2.4, "Lowest Priority Delivery Mode").
2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or INIT-deassert interrupt, or one of the MP protocol IPI messages (BIPI, FIPI, and SIPI), the interrupt is sent directly to the processor core for handling.
3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC looks for an open slot in one of its two pending interrupt queues contained in the IRR and ISR registers (see Figure 10-20). If a slot is available (see Section 10.8.4, "Interrupt Acceptance for Fixed Interrupts"), places the interrupt in the slot. If a slot is not available, it rejects the interrupt request and sends it back to the sender with a retry message.
4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 10.8.3.1, "Task and Processor Priorities").
5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC.

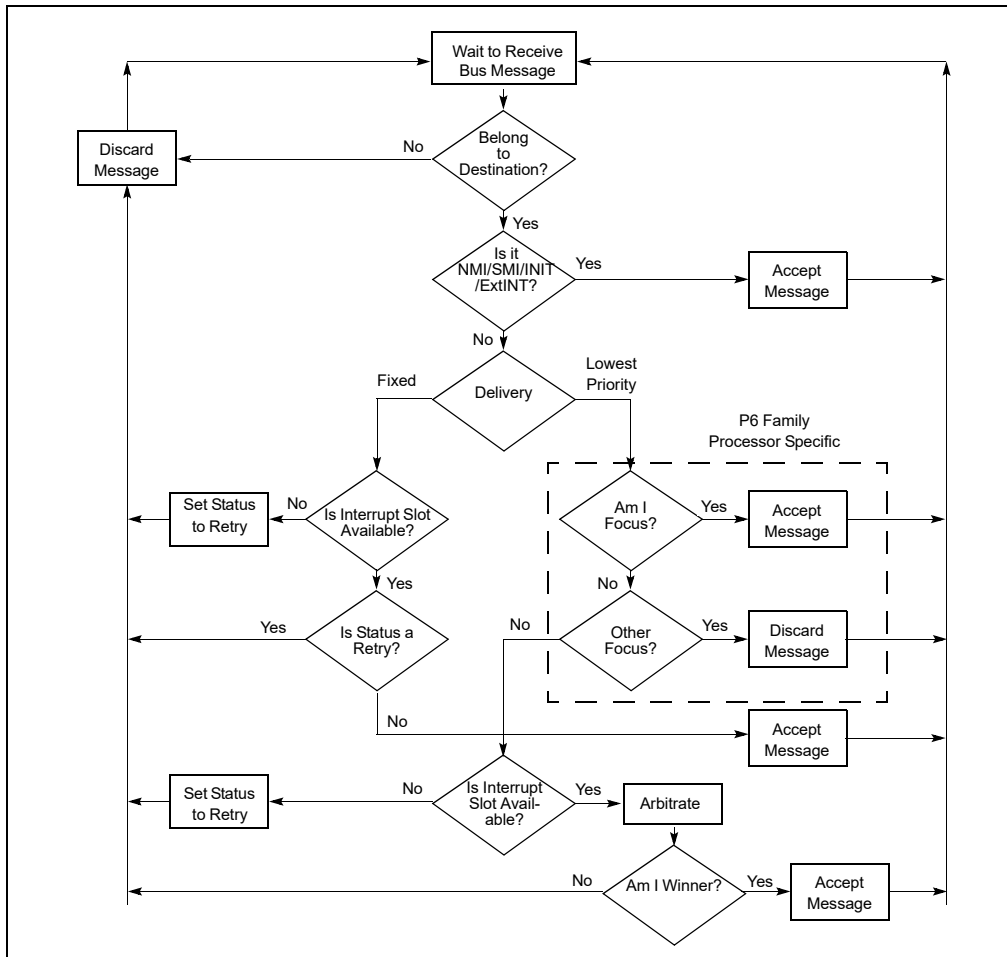


Figure 10-17. Interrupt Acceptance Flow Chart for the Local APIC (P6 Family and Pentium Processors)

register in the local APIC (see Section 10.8.5, “Signaling Interrupt Servicing Completion”). The act of writing to the EOI register causes the local APIC to delete the interrupt from its queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

The following sections describe the acceptance of interrupts and their handling by the local APIC and processor in greater detail.

10.8.3 Interrupt, Task, and Processor Priority

Each interrupt delivered to the processor through the local APIC has a priority based on its vector number. The local APIC uses this priority to determine when to service the interrupt relative to the other activities of the processor, including the servicing of other interrupts.

Each interrupt vector is an 8-bit value. The **interrupt-priority class** is the value of bits 7:4 of the interrupt vector. The lowest interrupt-priority class is 1 and the highest is 15; interrupts with vectors in the range 0–15 (with interrupt-priority class 0) are illegal and are never delivered. Because vectors 0–31 are reserved for dedicated uses by the Intel 64 and IA-32 architectures, software should configure interrupt vectors to use interrupt-priority classes in the range 2–15.

Each interrupt-priority class encompasses 16 vectors. The relative priority of interrupts within an interrupt-priority class is determined by the value of bits 3:0 of the vector number. The higher the value of those bits, the higher the

priority within that interrupt-priority class. Thus, each interrupt vector comprises two parts, with the high 4 bits indicating its interrupt-priority class and the low 4 bits indicating its ranking within the interrupt-priority class.

10.8.3.1 Task and Processor Priorities

The local APIC also defines a **task priority** and a **processor priority** that determine the order in which interrupts are handled. The **task-priority class** is the value of bits 7:4 of the task-priority register (TPR), which can be written by software (TPR is a read/write register); see Figure 10-18.

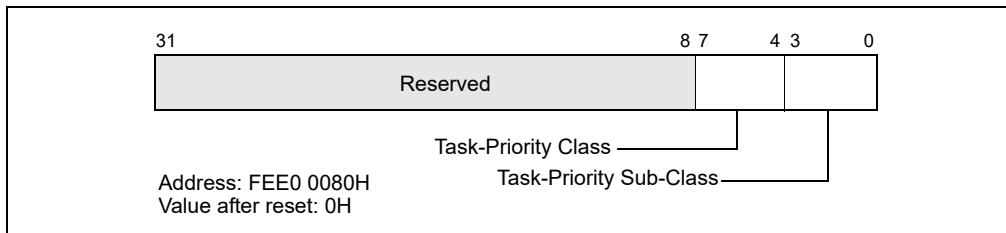


Figure 10-18. Task-Priority Register (TPR)

NOTE

In this discussion, the term “task” refers to a software defined task, process, thread, program, or routine that is dispatched to run on the processor by the operating system. It does not refer to an IA-32 architecture defined task as described in Chapter 7, “Task Management.”

The task priority allows software to set a priority threshold for interrupting the processor. This mechanism enables the operating system to temporarily block low priority interrupts from disturbing high-priority work that the processor is doing. The ability to block such interrupts using task priority results from the way that the TPR controls the value of the processor-priority register (PPR).⁷

The **processor-priority class** is a value in the range 0–15 that is maintained in bits 7:4 of the processor-priority register (PPR); see Figure 10-19. The PPR is a read-only register. The processor-priority class represents the current priority at which the processor is executing.

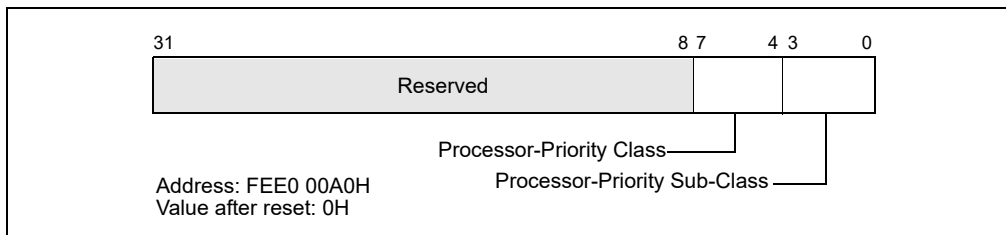


Figure 10-19. Processor-Priority Register (PPR)

The value of the PPR is based on the value of TPR and the value ISRV; ISRV is the vector number of the highest priority bit that is set in the ISR or 00H if no bit is set in the ISR. (See Section 10.8.4 for more details on the ISR.) The value of PPR is determined as follows:

- PPR[7:4] (the processor-priority class) the maximum of TPR[7:4] (the task- priority class) and ISRV[7:4] (the priority of the highest priority interrupt in service).
- PPR[3:0] (the processor-priority sub-class) is determined as follows:
 - If TPR[7:4] > ISRV[7:4], PPR[3:0] is TPR[3:0] (the task-priority sub-class).
 - If TPR[7:4] < ISRV[7:4], PPR[3:0] is 0.
 - If TPR[7:4] = ISRV[7:4], PPR[3:0] may be either TPR[3:0] or 0. The actual behavior is model-specific.

7. The TPR also determines the arbitration priority of the local processor; see Section 10.6.2.4, “Lowest Priority Delivery Mode.”

The processor-priority class determines the priority threshold for interrupting the processor. The processor will deliver only those interrupts that have an interrupt-priority class higher than the processor-priority class in the PPR. If the processor-priority class is 0, the PPR does not inhibit the delivery any interrupt; if it is 15, the processor inhibits the delivery of all interrupts. (The processor-priority mechanism does not affect the delivery of interrupts with the NMI, SMI, INIT, ExtINT, INIT-deassert, and start-up delivery modes.)

The processor does not use the processor-priority sub-class to determine which interrupts to delivery and which to inhibit. (The processor uses the processor-priority sub-class only to satisfy reads of the PPR.)

10.8.4 Interrupt Acceptance for Fixed Interrupts

The local APIC queues the fixed interrupts that it accepts in one of two interrupt pending registers: the interrupt request register (IRR) or in-service register (ISR). These two 256-bit read-only registers are shown in Figure 10-20. The 256 bits in these registers represent the 256 possible vectors; vectors 0 through 15 are reserved by the APIC (see also: Section 10.5.2, "Valid Interrupt Vectors").

NOTE

All interrupts with an NMI, SMI, INIT, ExtINT, start-up, or INIT-deassert delivery mode bypass the IRR and ISR registers and are sent directly to the processor core for servicing.

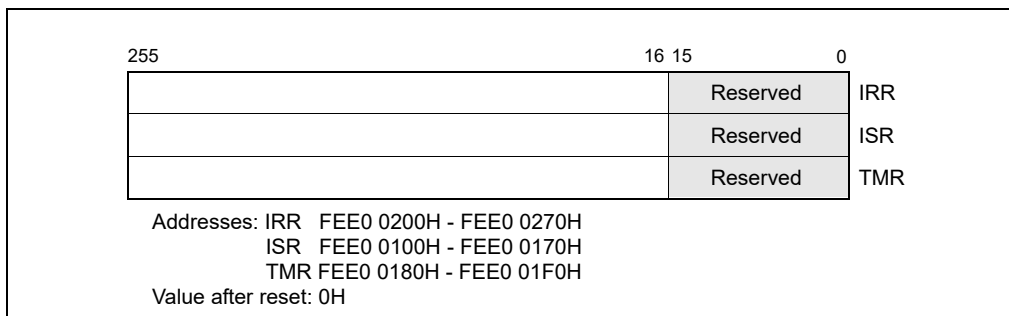


Figure 10-20. IRR, ISR and TMR Registers

The IRR contains the active interrupt requests that have been accepted, but not yet dispatched to the processor for servicing. When the local APIC accepts an interrupt, it sets the bit in the IRR that corresponds the vector of the accepted interrupt. When the processor core is ready to handle the next interrupt, the local APIC clears the highest priority IRR bit that is set and sets the corresponding ISR bit. The vector for the highest priority bit set in the ISR is then dispatched to the processor core for servicing.

While the processor is servicing the highest priority interrupt, the local APIC can send additional fixed interrupts by setting bits in the IRR. When the interrupt service routine issues a write to the EOI register (see Section 10.8.5, "Signaling Interrupt Servicing Completion"), the local APIC responds by clearing the highest priority ISR bit that is set. It then repeats the process of clearing the highest priority bit in the IRR and setting the corresponding bit in the ISR. The processor core then begins executing the service routing for the highest priority bit set in the ISR.

If more than one interrupt is generated with the same vector number, the local APIC can set the bit for the vector both in the IRR and the ISR. This means that for the Pentium 4 and Intel Xeon processors, the IRR and ISR can queue two interrupts for each interrupt vector: one in the IRR and one in the ISR. Any additional interrupts issued for the same interrupt vector are collapsed into the single bit in the IRR.

For the P6 family and Pentium processors, the IRR and ISR registers can queue no more than two interrupts per interrupt vector and will reject other interrupts that are received within the same vector.

If the local APIC receives an interrupt with an interrupt-priority class higher than that of the interrupt currently in service, and interrupts are enabled in the processor core, the local APIC dispatches the higher priority interrupt to the processor immediately (without waiting for a write to the EOI register). The currently executing interrupt handler is then interrupted so the higher-priority interrupt can be handled. When the handling of the higher-priority interrupt has been completed, the servicing of the interrupted interrupt is resumed.

The trigger mode register (TMR) indicates the trigger mode of the interrupt (see Figure 10-20). Upon acceptance of an interrupt into the IRR, the corresponding TMR bit is cleared for edge-triggered interrupts and set for level-triggered interrupts. If a TMR bit is set when an EOI cycle for its corresponding interrupt vector is generated, an EOI message is sent to all I/O APICs.

10.8.5 Signaling Interrupt Servicing Completion

For all interrupts except those delivered with the NMI, SMI, INIT, ExtINT, the start-up, or INIT-Deassert delivery mode, the interrupt handler must include a write to the end-of-interrupt (EOI) register (see Figure 10-21). This write must occur at the end of the handler routine, sometime before the IRET instruction. This action indicates that the servicing of the current interrupt is complete and the local APIC can issue the next interrupt from the ISR.

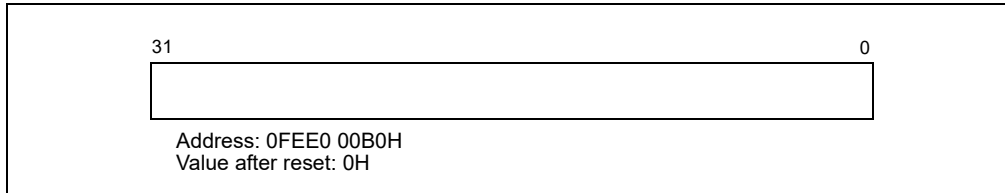


Figure 10-21. EOI Register

Upon receiving an EOI, the APIC clears the highest priority bit in the ISR and dispatches the next highest priority interrupt to the processor. If the terminated interrupt was a level-triggered interrupt, the local APIC also sends an end-of-interrupt message to all I/O APICs.

System software may prefer to direct EOIs to specific I/O APICs rather than having the local APIC send end-of-interrupt messages to all I/O APICs.

Software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register (see Section 10.9). If this bit is set, a broadcast EOI is not generated on an EOI cycle even if the associated TMR bit indicates that the current interrupt was level-triggered. The default value for the bit is 0, indicating that EOI broadcasts are performed.

Bit 12 of the Spurious Interrupt Vector Register is reserved to 0 if the processor does not support suppression of EOI broadcasts. Support for EOI-broadcast suppression is reported in bit 24 in the Local APIC Version Register (see Section 10.4.8); the feature is supported if that bit is set to 1. When supported, the feature is available in both xAPIC mode and x2APIC mode.

System software desiring to perform directed EOIs for level-triggered interrupts should set bit 12 of the Spurious Interrupt Vector Register and follow each the EOI to the local xAPIC for a level triggered interrupt with a directed EOI to the I/O APIC generating the interrupt (this is done by writing to the I/O APIC's EOI register). System software performing directed EOIs must retain a mapping associating level-triggered interrupts with the I/O APICs in the system.

10.8.6 Task Priority in IA-32e Mode

In IA-32e mode, operating systems can manage the 16 interrupt-priority classes (see Section 10.8.3, "Interrupt, Task, and Processor Priority") explicitly using the task priority register (TPR). Operating systems can use the TPR to temporarily block specific (low-priority) interrupts from interrupting a high-priority task. This is done by loading TPR with a value in which the task-priority class corresponds to the highest interrupt-priority class that is to be blocked. For example:

- Loading the TPR with a task-priority class of 8 (01000B) blocks all interrupts with an interrupt-priority class of 8 or less while allowing all interrupts with an interrupt-priority class of 9 or more to be recognized.
- Loading the TPR with a task-priority class of 0 enables all external interrupts.
- Loading the TPR with a task-priority class of 0FH (01111B) disables all external interrupts.

The TPR (shown in Figure 10-18) is cleared to 0 on reset. In 64-bit mode, software can read and write the TPR using an alternate interface, MOV CR8 instruction. The new task-priority class is established when the MOV CR8

instruction completes execution. Software does not need to force serialization after loading the TPR using MOV CR8.

Use of the MOV CRn instruction requires a privilege level of 0. Programs running at privilege level greater than 0 cannot read or write the TPR. An attempt to do so causes a general-protection exception. The TPR is abstracted from the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external device, such as an APIC or 8259. Typically, the IC provides a priority mechanism similar or identical to the TPR. The IC, however, is considered implementation-dependent with the under-lying priority mechanisms subject to change. CR8, by contrast, is part of the Intel 64 architecture. Software can depend on this definition remaining unchanged.

Figure 10-22 shows the layout of CR8; only the low four bits are used. The remaining 60 bits are reserved and must be written with zeros. Failure to do this causes a general-protection exception.

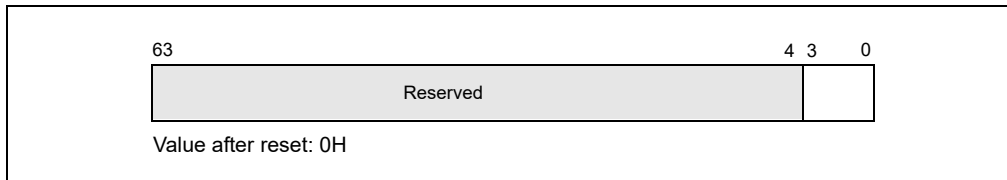


Figure 10-22. CR8 Register

10.8.6.1 Interaction of Task Priorities between CR8 and APIC

The first implementation of Intel 64 architecture includes a local advanced programmable interrupt controller (APIC) that is similar to the APIC used with previous IA-32 processors. Some aspects of the local APIC affect the operation of the architecturally defined task priority register and the programming interface using CR8.

Notable CR8 and APIC interactions are:

- The processor powers up with the local APIC enabled.
- The APIC must be enabled for CR8 to function as the TPR. Writes to CR8 are reflected into the APIC Task Priority Register.
- APIC.TPR[bits 7:4] = CR8[bits 3:0], APIC.TPR[bits 3:0] = 0. A read of CR8 returns a 64-bit value which is the value of TPR[bits 7:4], zero extended to 64 bits.

There are no ordering mechanisms between direct updates of the APIC.TPR and CR8. Operating software should implement either direct APIC TPR updates or CR8 style TPR updates but not mix them. Software can use a serializing instruction (for example, CPUID) to serialize updates between MOV CR8 and stores to the APIC.

10.9 SPURIOUS INTERRUPT

A special situation may occur when a processor raises its task priority to be greater than or equal to the level of the interrupt for which the processor INTR signal is currently being asserted. If at the time the INTA cycle is issued, the interrupt that was to be dispensed has become masked (programmed by software), the local APIC will deliver a spurious-interrupt vector. Dispensing the spurious-interrupt vector does not affect the ISR, so the handler for this vector should return without an EOI.

The vector number for the spurious-interrupt vector is specified in the spurious-interrupt vector register (see Figure 10-23). The functions of the fields in this register are as follows:

- Spurious Vector** Determines the vector number to be delivered to the processor when the local APIC generates a spurious vector.
- (Pentium 4 and Intel Xeon processors.) Bits 0 through 7 of the this field are programmable by software.
 - (P6 family and Pentium processors). Bits 4 through 7 of the this field are programmable by software, and bits 0 through 3 are hardwired to logical ones. Software writes to bits 0 through 3 have no effect.

APIC Software Enable/Disable

Allows software to temporarily enable (1) or disable (0) the local APIC (see Section 10.4.3, "Enabling or Disabling the Local APIC").

Focus Processor Checking

Determines if focus processor checking is enabled (0) or disabled (1) when using the lowest-priority delivery mode. In Pentium 4 and Intel Xeon processors, this bit is reserved and should be cleared to 0.

Suppress EOI Broadcasts

Determines whether an EOI for a level-triggered interrupt causes EOI messages to be broadcast to the I/O APICs (0) or not (1). See Section 10.8.5. The default value for this bit is 0, indicating that EOI broadcasts are performed. This bit is reserved to 0 if the processor does not support EOI-broadcast suppression.

NOTE

Do not program an LVT or IOAPIC RTE with a spurious vector even if you set the mask bit. A spurious vector ISR does not do an EOI. If for some reason an interrupt is generated by an LVT or RTE entry, the bit in the in-service register will be left set for the spurious vector. This will mask all interrupts at the same or lower priority

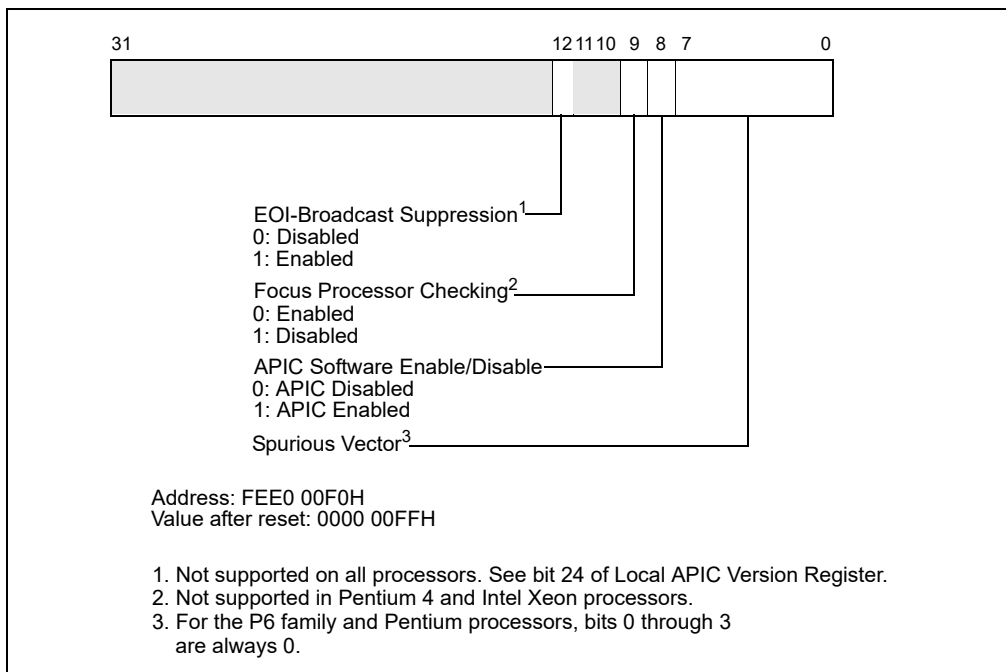


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

10.10 APIC BUS MESSAGE PASSING MECHANISM AND PROTOCOL (P6 FAMILY, PENTIUM PROCESSORS)

The Pentium 4 and Intel Xeon processors pass messages among the local and I/O APICs on the system bus, using the system bus message passing mechanism and protocol.

The P6 family and Pentium processors, pass messages among the local and I/O APICs on the serial APIC bus, as follows. Because only one message can be sent at a time on the APIC bus, the I/O APIC and local APICs employ a "rotating priority" arbitration protocol to gain permission to send a message on the APIC bus. One or more APICs may start sending their messages simultaneously. At the beginning of every message, each APIC presents the type of the message it is sending and its current arbitration priority on the APIC bus. This information is used for arbitration. After each arbitration cycle (within an arbitration round), only the potential winners keep driving the bus.

By the time all arbitration cycles are completed, there will be only one APIC left driving the bus. Once a winner is selected, it is granted exclusive use of the bus, and will continue driving the bus to send its actual message.

After each successfully transmitted message, all APICs increase their arbitration priority by 1. The previous winner (that is, the one that has just successfully transmitted its message) assumes a priority of 0 (lowest). An agent whose arbitration priority was 15 (highest) during arbitration, but did not send a message, adopts the previous winner's arbitration priority, incremented by 1.

Note that the arbitration protocol described above is slightly different if one of the APICs issues a special End-Of-Interrupt (EOI). This high-priority message is granted the bus regardless of its sender's arbitration priority, unless more than one APIC issues an EOI message simultaneously. In the latter case, the APICs sending the EOI messages arbitrate using their arbitration priorities.

If the APICs are set up to use "lowest priority" arbitration (see Section 10.6.2.4, "Lowest Priority Delivery Mode") and multiple APICs are currently executing at the lowest priority (the value in the APR register), the arbitration priorities (unique values in the Arb ID register) are used to break ties. All 8 bits of the APR are used for the lowest priority arbitration.

10.10.1 Bus Message Formats

See Section 10.13, "APIC Bus Message Formats," for a description of bus message formats used to transmit messages on the serial APIC bus.

10.11 MESSAGE SIGNALLED INTERRUPTS

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

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

The capabilities mechanism provided by the *PCI Local Bus Specification* is used to identify and configure MSI capable PCI devices. Among other fields, this structure contains a Message Data Register and a Message Address Register. To request service, the PCI device function writes the contents of the Message Data Register to the address contained in the Message Address Register (and the Message Upper Address register for 64-bit message addresses).

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

10.11.1 Message Address Register Format

The format of the Message Address Register (lower 32-bits) is shown in Figure 10-24.

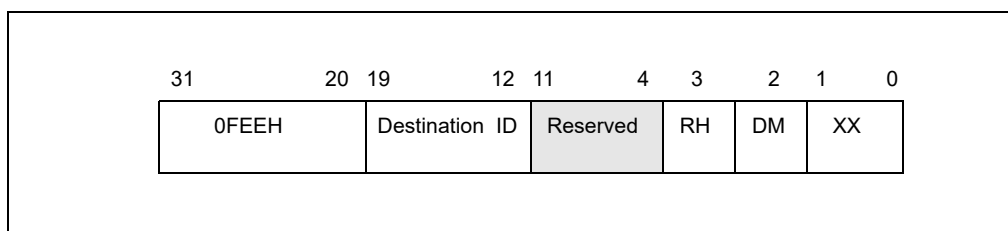


Figure 10-24. Layout of the MSI Message Address Register

Fields in the Message Address Register are as follows:

1. **Bits 31-20** — These bits contain a fixed value for interrupt messages (0FEEH). This value locates interrupts at the 1-MByte area with a base address of 4G – 18M. All accesses to this region are directed as interrupt messages. Care must be taken to ensure that no other device claims the region as I/O space.
2. **Destination ID** — This field contains an 8-bit destination ID. It identifies the message's target processor(s). The destination ID corresponds to bits 63:56 of the I/O APIC Redirection Table Entry if the IOAPIC is used to dispatch the interrupt to the processor(s).
3. **Redirection hint indication (RH)** — When this bit is set, the message is directed to the processor with the lowest interrupt priority among processors that can receive the interrupt.
 - When RH is 0, the interrupt is directed to the processor listed in the Destination ID field.
 - When RH is 1 and the physical destination mode is used, the Destination ID field must not be set to FFH; it must point to a processor that is present and enabled to receive the interrupt.
 - When RH is 1 and the logical destination mode is active in a system using a flat addressing model, the Destination ID field must be set so that bits set to 1 identify processors that are present and enabled to receive the interrupt.
 - If RH is set to 1 and the logical destination mode is active in a system using cluster addressing model, then Destination ID field must not be set to FFH; the processors identified with this field must be present and enabled to receive the interrupt.
4. **Destination mode (DM)** — This bit indicates whether the Destination ID field should be interpreted as logical or physical APIC ID for delivery of the lowest priority interrupt.
 - If RH is 1 and DM is 0, the Destination ID field is in physical destination mode and only the processor in the system that has the matching APIC ID is considered for delivery of that interrupt (this means no redirection).
 - If RH is 1 and DM is 1, the Destination ID Field is interpreted as in logical destination mode and the redirection is limited to only those processors that are part of the logical group of processors based on the processor's logical APIC ID and the Destination ID field in the message. The logical group of processors consists of those identified by matching the 8-bit Destination ID with the logical destination identified by the Destination Format Register and the Logical Destination Register in each local APIC. The details are similar to those described in Section 10.6.2, "Determining IPI Destination."
 - If RH is 0, then the DM bit is ignored and the message is sent ahead independent of whether the physical or logical destination mode is used.

10.11.2 Message Data Register Format

The layout of the Message Data Register is shown in Figure 10-25.

Reserved fields are not assumed to be any value. Software must preserve their contents on writes. Other fields in the Message Data Register are described below.

1. **Vector** — This 8-bit field contains the interrupt vector associated with the message. Values range from 010H to 0FEH. Software must guarantee that the field is not programmed with vector 00H to 0FH.
2. **Delivery Mode** — This 3-bit field specifies how the interrupt receipt is handled. Delivery Modes operate only in conjunction with specified Trigger Modes. Correct Trigger Modes must be guaranteed by software. Restrictions are indicated below:
 - a. **000B (Fixed Mode)** — Deliver the signal to all the agents listed in the destination. The Trigger Mode for fixed delivery mode can be edge or level.
 - b. **001B (Lowest Priority)** — Deliver the signal to the agent that is executing at the lowest priority of all agents listed in the destination field. The trigger mode can be edge or level.
 - c. **010B (System Management Interrupt or SMI)** — The delivery mode is edge only. For systems that rely on SMI semantics, the vector field is ignored but must be programmed to all zeroes for future compatibility.

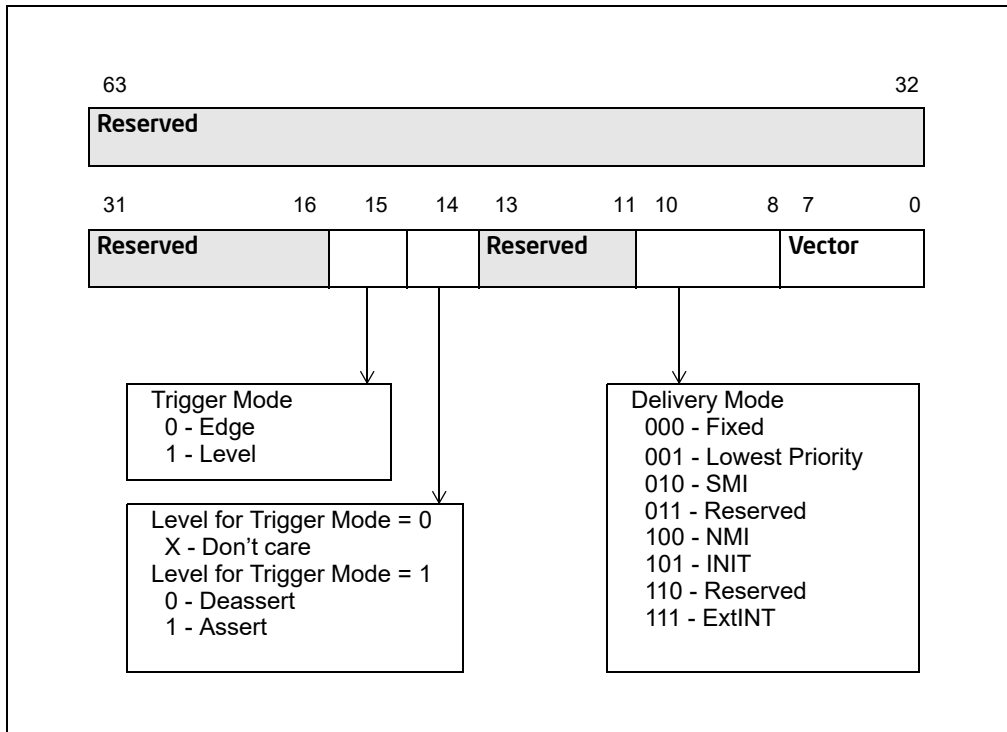


Figure 10-25. Layout of the MSI Message Data Register

- d. **100B (NMI)** — Deliver the signal to all the agents listed in the destination field. The vector information is ignored. NMI is an edge triggered interrupt regardless of the Trigger Mode Setting.
 - e. **101B (INIT)** — Deliver this signal to all the agents listed in the destination field. The vector information is ignored. INIT is an edge triggered interrupt regardless of the Trigger Mode Setting.
 - f. **111B (ExtINT)** — Deliver the signal to the INTR signal of all agents in the destination field (as an interrupt that originated from an 8259A compatible interrupt controller). The vector is supplied by the INTA cycle issued by the activation of the ExtINT. ExtINT is an edge triggered interrupt.
3. **Level** — Edge triggered interrupt messages are always interpreted as assert messages. For edge triggered interrupts this field is not used. For level triggered interrupts, this bit reflects the state of the interrupt input.
 4. **Trigger Mode** — This field indicates the signal type that will trigger a message.
 - a. **0** — Indicates edge sensitive.
 - b. **1** — Indicates level sensitive.

10.12 EXTENDED XAPIC (X2APIC)

The x2APIC architecture extends the xAPIC architecture (described in Section 10.4) in a backward compatible manner and provides forward extendability for future Intel platform innovations. Specifically, the x2APIC architecture does the following.

- Retains all key elements of compatibility to the xAPIC architecture.
 - Delivery modes.
 - Interrupt and processor priorities.
 - Interrupt sources.
 - Interrupt destination types.
- Provides extensions to scale processor addressability for both the logical and physical destination modes.

- Adds new features to enhance performance of interrupt delivery.
- Reduces complexity of logical destination mode interrupt delivery on link based platform architectures.
- Uses MSR programming interface to access APIC registers in x2APIC mode instead of memory-mapped interfaces. Memory-mapped interface is supported when operating in xAPIC mode.

10.12.1 Detecting and Enabling x2APIC Mode

Processor support for x2APIC mode can be detected by executing CPUID with EAX=1 and then checking ECX, bit 21 ECX. If CPUID.(EAX=1):ECX.21 is set, the processor supports the x2APIC capability and can be placed into the x2APIC mode.

System software can place the local APIC in the x2APIC mode by setting the x2APIC mode enable bit (bit 10) in the IA32_APIC_BASE MSR at MSR address 01BH. The layout for the IA32_APIC_BASE MSR is shown in Figure 10-26.

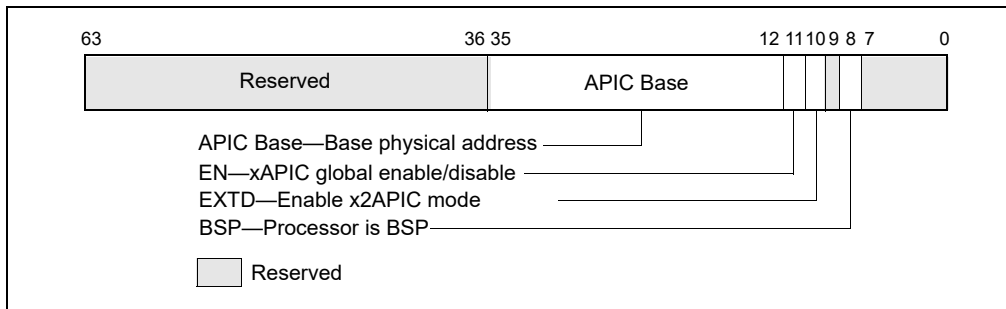


Figure 10-26. IA32_APIC_BASE MSR Supporting x2APIC

Table 10-5, “x2APIC operating mode configurations” describe the possible combinations of the enable bit (EN - bit 11) and the extended mode bit (EXTD - bit 10) in the IA32_APIC_BASE MSR.

Table 10-5. x2APIC Operating Mode Configurations

xAPIC global enable (IA32_APIC_BASE[11])	x2APIC enable (IA32_APIC_BASE[10])	Description
0	0	local APIC is disabled
0	1	Invalid
1	0	local APIC is enabled in xAPIC mode
1	1	local APIC is enabled in x2APIC mode

Once the local APIC has been switched to x2APIC mode (EN = 1, EXT D = 1), switching back to xAPIC mode would require system software to disable the local APIC unit. Specifically, attempting to write a value to the IA32_APIC_BASE MSR that has (EN= 1, EXT D = 0) when the local APIC is enabled and in x2APIC mode causes a general-protection exception. Once bit 10 in IA32_APIC_BASE MSR is set, the only way to leave x2APIC mode using IA32_APIC_BASE would require a WRMSR to set both bit 11 and bit 10 to zero. Section 10.12.5, “x2APIC State Transitions” provides a detailed state diagram for the state transitions allowed for the local APIC.

10.12.1.1 Instructions to Access APIC Registers

In x2APIC mode, system software uses RDMSR and WRMSR to access the APIC registers. The MSR addresses for accessing the x2APIC registers are architecturally defined and specified in Section 10.12.1.2, “x2APIC Register Address Space”. Executing the RDMSR instruction with the APIC register address specified in ECX returns the content of bits 0 through 31 of the APIC registers in EAX. Bits 32 through 63 are returned in register EDX - these bits are reserved if the APIC register being read is a 32-bit register. Similarly executing the WRMSR instruction with the APIC register address in ECX, writes bits 0 to 31 of register EAX to bits 0 to 31 of the specified APIC register. If the register is a 64-bit register then bits 0 to 31 of register EDX are written to bits 32 to 63 of the APIC register. The

Interrupt Command Register is the only APIC register that is implemented as a 64-bit MSR. The semantics of handling reserved bits are defined in Section 10.12.1.3, "Reserved Bit Checking".

10.12.1.2 x2APIC Register Address Space

The MSR address range 800H through 8FFH is architecturally reserved and dedicated for accessing APIC registers in x2APIC mode. Table 10-6 lists the APIC registers that are available in x2APIC mode. When appropriate, the table also gives the offset at which each register is available on the page referenced by IA32_APIC_BASE[35:12] in xAPIC mode.

There is a one-to-one mapping between the x2APIC MSRs and the legacy xAPIC register offsets with the following exceptions:

- The Destination Format Register (DFR): The DFR, supported at offset 0E0H in xAPIC mode, is not supported in x2APIC mode. There is no MSR with address 80EH.
- The Interrupt Command Register (ICR): The two 32-bit registers in xAPIC mode (at offsets 300H and 310H) are merged into a single 64-bit MSR in x2APIC mode (with MSR address 830H). There is no MSR with address 831H.
- The SELF IPI register. This register is available only in x2APIC mode at address 83FH. In xAPIC mode, there is no register defined at offset 3F0H.

MSR addresses in the range 800H–8FFH that are not listed in Table 10-6 (including 80EH and 831H) are reserved. Executions of RDMSR and WRMSR that attempt to access such addresses cause general-protection exceptions.

The MSR address space is compressed to allow for future growth. Every 32 bit register on a 128-bit boundary in the legacy MMIO space is mapped to a single MSR in the local x2APIC MSR address space. The upper 32-bits of all x2APIC MSRs (except for the ICR) are reserved.

Table 10-6. Local APIC Register Address Map Supported by x2APIC

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
802H	020H	Local APIC ID register	Read-only ¹	See Section 10.12.5.1 for initial values.
803H	030H	Local APIC Version register	Read-only	Same version used in xAPIC mode and x2APIC mode.
808H	080H	Task Priority Register (TPR)	Read/write	Bits 31:8 are reserved. ²
80AH	0A0H	Processor Priority Register (PPR)	Read-only	
80BH	0B0H	EOI register	Write-only ³	WRMSR of a non-zero value causes #GP(0).
80DH	0D0H	Logical Destination Register (LDR)	Read-only	Read/write in xAPIC mode.
80FH	0F0H	Spurious Interrupt Vector Register (SVR)	Read/write	See Section 10.9 for reserved bits.
810H	100H	In-Service Register (ISR); bits 31:0	Read-only	
811H	110H	ISR bits 63:32	Read-only	
812H	120H	ISR bits 95:64	Read-only	
813H	130H	ISR bits 127:96	Read-only	
814H	140H	ISR bits 159:128	Read-only	
815H	150H	ISR bits 191:160	Read-only	
816H	160H	ISR bits 223:192	Read-only	

Table 10-6. Local APIC Register Address Map Supported by x2APIC (Contd.)

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
817H	170H	ISR bits 255:224	Read-only	
818H	180H	Trigger Mode Register (TMR); bits 31:0	Read-only	
819H	190H	TMR bits 63:32	Read-only	
81AH	1A0H	TMR bits 95:64	Read-only	
81BH	1B0H	TMR bits 127:96	Read-only	
81CH	1C0H	TMR bits 159:128	Read-only	
81DH	1D0H	TMR bits 191:160	Read-only	
81EH	1E0H	TMR bits 223:192	Read-only	
81FH	1F0H	TMR bits 255:224	Read-only	
820H	200H	Interrupt Request Register (IRR); bits 31:0	Read-only	
821H	210H	IRR bits 63:32	Read-only	
822H	220H	IRR bits 95:64	Read-only	
823H	230H	IRR bits 127:96	Read-only	
824H	240H	IRR bits 159:128	Read-only	
825H	250H	IRR bits 191:160	Read-only	
826H	260H	IRR bits 223:192	Read-only	
827H	270H	IRR bits 255:224	Read-only	
828H	280H	Error Status Register (ESR)	Read/write	WRMSR of a non-zero value causes #GP(0). See Section 10.5.3.
82FH	2F0H	LVT CMCI register	Read/write	See Figure 10-8 for reserved bits.
830H ⁴	300H and 310H	Interrupt Command Register (ICR)	Read/write	See Figure 10-28 for reserved bits
832H	320H	LVT Timer register	Read/write	See Figure 10-8 for reserved bits.
833H	330H	LVT Thermal Sensor register	Read/write	See Figure 10-8 for reserved bits.
834H	340H	LVT Performance Monitoring register	Read/write	See Figure 10-8 for reserved bits.
835H	350H	LVT LINT0 register	Read/write	See Figure 10-8 for reserved bits.
836H	360H	LVT LINT1 register	Read/write	See Figure 10-8 for reserved bits.
837H	370H	LVT Error register	Read/write	See Figure 10-8 for reserved bits.
838H	380H	Initial Count register (for Timer)	Read/write	
839H	390H	Current Count register (for Timer)	Read-only	
83EH	3E0H	Divide Configuration Register (DCR; for Timer)	Read/write	See Figure 10-10 for reserved bits.
83FH	Not available	SELF IPI ⁵	Write-only	Available only in x2APIC mode.

NOTES:

1. WRMSR causes #GP(0) for read-only registers.

2. WRMSR causes #GP(0) for attempts to set a reserved bit to 1 in a read/write register (including bits 63:32 of each register).
3. RDMSR causes #GP(0) for write-only registers.
4. MSR 831H is reserved; read/write operations cause general-protection exceptions. The contents of the APIC register at MMIO offset 310H are accessible in x2APIC mode through the MSR at address 830H.
5. SELF IPI register is supported only in x2APIC mode.

10.12.1.3 Reserved Bit Checking

Section 10.12.1.2 and Table 10-6 specifies the reserved bit definitions for the APIC registers in x2APIC mode. Non-zero writes (by WRMSR instruction) to reserved bits to these registers will raise a general protection fault exception while reads return zeros (RsvdZ semantics).

In x2APIC mode, the local APIC ID register is increased to 32 bits wide. This enables $2^{32}-1$ processors to be addressable in physical destination mode. This 32-bit value is referred to as “x2APIC ID”. A processor implementation may choose to support less than 32 bits in its hardware. System software should be agnostic to the actual number of bits that are implemented. All non-implemented bits will return zeros on reads by software.

The APIC ID value of FFFF_FFFFH and the highest value corresponding to the implemented bit-width of the local APIC ID register in the system are reserved and cannot be assigned to any logical processor.

In x2APIC mode, the local APIC ID register is a read-only register to system software and will be initialized by hardware. It is accessed via the RDMSR instruction reading the MSR at address 0802H.

Each logical processor in the system (including clusters with a communication fabric) must be configured with a unique x2APIC ID to avoid collisions of x2APIC IDs. On DP and high-end MP processors targeted to specific market segments and depending on the system configuration, it is possible that logical processors in different and “un-connected” clusters power up initialized with overlapping x2APIC IDs. In these configurations, a model-specific means may be provided in those product segments to enable BIOS and/or platform firmware to re-configure the x2APIC IDs in some clusters to provide for unique and non-overlapping system wide IDs before configuring the disconnected components into a single system.

10.12.2 x2APIC Register Availability

The local APIC registers can be accessed via the MSR interface only when the local APIC has been switched to the x2APIC mode as described in Section 10.12.1. Accessing any APIC register in the MSR address range 0800H through 08FFH via RDMSR or WRMSR when the local APIC is not in x2APIC mode causes a general-protection exception. In x2APIC mode, the memory mapped interface is not available and any access to the MMIO interface will behave similar to that of a legacy xAPIC in globally disabled state. Table 10-7 provides the interactions between the legacy & extended modes and the legacy and register interfaces.

Table 10-7. MSR/MMIO Interface of a Local x2APIC in Different Modes of Operation

	MMIO Interface	MSR Interface
xAPIC mode	Available	General-protection exception
x2APIC mode	Behavior identical to xAPIC in globally disabled state	Available

10.12.3 MSR Access in x2APIC Mode

To allow for efficient access to the APIC registers in x2APIC mode, the serializing semantics of WRMSR are relaxed when writing to the APIC registers. Thus, system software should not use “WRMSR to APIC registers in x2APIC mode” as a serializing instruction. Read and write accesses to the APIC registers will occur in program order. A WRMSR to an APIC register may complete before all preceding stores are globally visible; software can prevent this by inserting a serializing instruction or the sequence MFENCE;LFENCE before the WRMSR.

The RDMSR instruction is not serializing and this behavior is unchanged when reading APIC registers in x2APIC mode. System software accessing the APIC registers using the RDMSR instruction should not expect a serializing behavior. (Note: The MMIO-based xAPIC interface is mapped by system software as an un-cached region. Consequently, read/writes to the xAPIC-MMIO interface have serializing semantics in the xAPIC mode.)

10.12.4 VM-Exit Controls for MSRs and x2APIC Registers

The VMX architecture allows a VMM to specify lists of MSRs to be loaded or stored on VMX transitions using the VMX-transition MSR areas (see VM-exit MSR-store address field, VM-exit MSR-load address field, and VM-entry MSR-load address field in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).

The X2APIC MSRs cannot to be loaded and stored on VMX transitions. A VMX transition fails if the VMM has specified that the transition should access any MSRs in the address range from 0000_0800H to 0000_08FFH (the range used for accessing the X2APIC registers). Specifically, processing of a 128-bit entry in any of the VMX-transition MSR areas fails if bits 31:0 of that entry (represented as ENTRY_LOW_DW) satisfies the expression: "ENTRY_LOW_DW & FFFF800H = 0000800H". Such a failure causes an associated VM entry to fail (by reloading host state) and causes an associated VM exit to lead to VMX abort.

10.12.5 x2APIC State Transitions

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and reset.

10.12.5.1 x2APIC States

The valid states for a local x2APIC unit are listed in Table 10-5.

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0.
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0.
- x2APIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=1.
- Invalid: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=1.

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. An execution of WRMSR to the IA32_APIC_BASE_MSR that attempts a transition from a valid state to this invalid state causes a general-protection exception. Figure 10-27 shows the comprehensive state transition diagram for a local x2APIC unit.

On coming out of reset, the local APIC unit is enabled and is in the xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0. The APIC registers are initialized as follows.

- The local APIC ID is initialized by hardware with a 32 bit ID (x2APIC ID). The lowest 8 bits of the x2APIC ID are the legacy local xAPIC ID, and are stored in the upper 8 bits of the APIC register for access in xAPIC mode.
- The following APIC registers are reset to all zeros for those fields that are defined in the xAPIC mode.
 - IRR, ISR, TMR, ICR, LDR, TPR, Divide Configuration Register (See Section 10.4 through Section 10.6 for details of individual APIC registers).
 - Timer initial count and timer current count registers.
- The LVT registers are reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The Spurious Interrupt Vector Register is initialized to 000000FFH.
- The DFR (available only in xAPIC mode) is reset to all 1s.
- SELF IPI register is reset to zero.

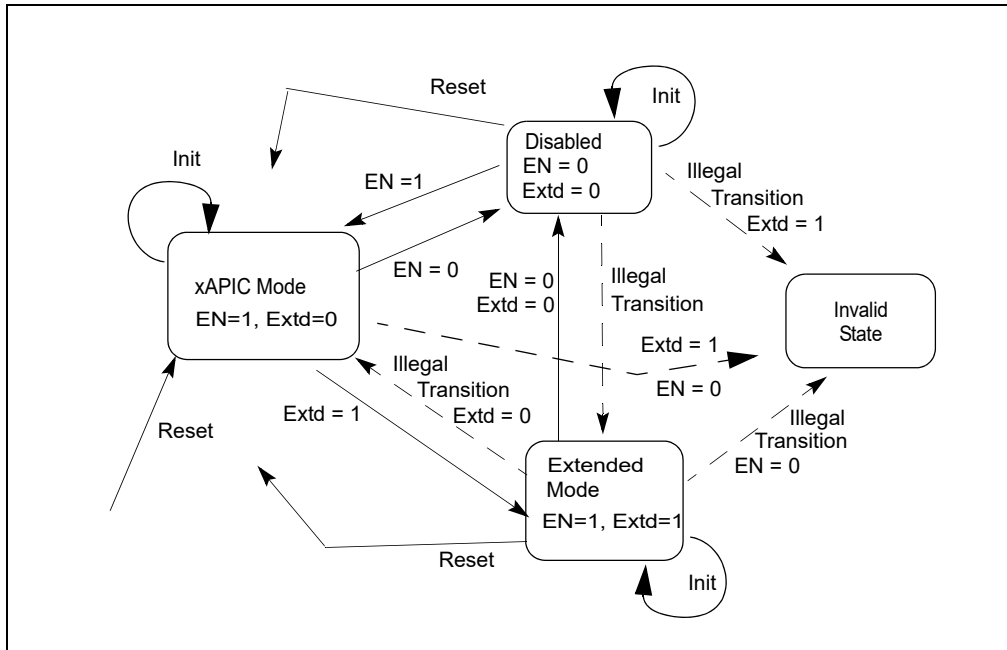


Figure 10-27. Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and Reset

x2APIC After Reset

The valid transitions from the xAPIC mode state are:

- to the x2APIC mode by setting EXT to 1 (resulting EN=1, EXTD= 1). The physical x2APIC ID (see Figure 10-6) is preserved across this transition and the logical x2APIC ID (see Figure 10-29) is initialized by hardware during this transition as documented in Section 10.12.10.2. The state of the extended fields in other APIC registers, which was not initialized at reset, is not architecturally defined across this transition and system software should explicitly initialize those programmable APIC registers.
- to the disabled state by setting EN to 0 (resulting EN=0, EXTD= 0).

The result of an INIT in the xAPIC state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is preserved (the 8-bit xAPIC ID is in the upper 8 bits of the APIC ID register). All the other APIC registers are initialized as a result of INIT.

A reset in this state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is initialized as described in Section 10.12.5.1. All the other APIC registers are initialized described in Section 10.12.5.1.

x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid, and the corresponding WRMSR to the IA32_APIC_BASE MSR causes a general-protection exception.

A reset in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 10.12.5.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.

x2APIC Transitions From Disabled Mode

From the disabled state, the only valid x2APIC transition using IA32_APIC_BASE is to the xAPIC mode (EN= 1, EXTD = 0). Thus the only means to transition from x2APIC mode to xAPIC mode is a two-step process:

- first transition from x2APIC mode to local APIC disabled mode (EN= 0, EXTD = 0),
- followed by another transition from disabled mode to xAPIC mode (EN= 1, EXTD= 0).

Consequently, all the APIC register states in the x2APIC, except for the x2APIC ID (32 bits), are not preserved across mode transitions.

A reset in the disabled state places the x2APIC in the xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 10.12.5.1.

An INIT in the disabled state keeps the x2APIC in the disabled state.

State Changes From xAPIC Mode to x2APIC Mode

After APIC register states have been initialized by software in xAPIC mode, a transition from xAPIC mode to x2APIC mode does not affect most of the APIC register states, except the following:

- The Logical Destination Register is not preserved.
- Any APIC ID value written to the memory-mapped local APIC ID register is not preserved.
- The high half of the Interrupt Command Register is not preserved.

10.12.6 Routing of Device Interrupts in x2APIC Mode

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

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

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

10.12.7 Initialization by System Software

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

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

10.12.8 CPUID Extensions And Topology Enumeration

For Intel 64 and IA-32 processors that support x2APIC, a value of 1 reported by CPUID.01H:ECX[21] indicates that the processor supports x2APIC and the extended topology enumeration leaf (CPUID.0BH).

The extended topology enumeration leaf can be accessed by executing CPUID with EAX = 0BH. Processors that do not support x2APIC may support CPUID leaf 0BH. Software can detect the availability of the extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal to 11 (0BH), then proceed to next step
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. If available, the extended topology enumeration leaf is the preferred mechanism for enumerating topology. The presence of CPUID leaf 0BH in a processor does not guarantee support for x2APIC. If CPUID.EAX=0BH, ECX=0H:EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

The extended topology enumeration leaf is intended to assist software with enumerating processor topology on systems that requires 32-bit x2APIC IDs to address individual logical processors. Details of CPUID leaf 0BH can be found in the reference pages of CPUID in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Processor topology enumeration algorithm for processors supporting the extended topology enumeration leaf of CPUID and processors that do not support CPUID leaf 0BH are treated in Section 8.9.4, "Algorithm for Three-Level Mappings of APIC_ID".

10.12.8.1 Consistency of APIC IDs and CPUID

The consistency of physical x2APIC ID in MSR 802H in x2APIC mode and the 32-bit value returned in CPUID.0BH:EDX is facilitated by processor hardware.

CPUID.0BH:EDX will report the full 32 bit ID, in xAPIC and x2APIC mode. This allows BIOS to determine if a system has processors with IDs exceeding the 8-bit initial APIC ID limit (CPUID.01H:EBX[31:24]). Initial APIC ID (CPUID.01H:EBX[31:24]) is always equal to CPUID.0BH:EDX[7:0].

If the values of CPUID.0BH:EDX reported by all logical processors in a system are less than 255, BIOS can transfer control to OS in xAPIC mode.

If the values of CPUID.0BH:EDX reported by some logical processors in a system are greater than or equal to 255, BIOS must support two options to hand off to OS.

- If BIOS enables logical processors with x2APIC IDs greater than 255, then it should enable x2APIC in the Boot Strap Processor (BSP) and all Application Processors (AP) before passing control to the OS. Applications requiring processor topology information must use OS provided services based on x2APIC IDs or CPUID.0BH leaf.
- If a BIOS transfers control to OS in xAPIC mode, then the BIOS must ensure that only logical processors with CPUID.0BH:EDX value less than 255 are enabled. BIOS initialization on all logical processors with CPUID.0B:EDX values greater than or equal to 255 must (a) disable APIC and execute CLI in each logical processor, and (b) leave these logical processor in the lowest power state so that these processors do not respond to INIT IPI during OS boot. The BSP and all the enabled logical processor operate in xAPIC mode after BIOS passed control to OS. Application requiring processor topology information can use OS provided legacy services based on 8-bit initial APIC IDs or legacy topology information from CPUID.01H and CPUID 04H leaves. Even if the BIOS passes control in xAPIC mode, an OS can switch the processors to x2APIC mode later. BIOS SMM handler should always read the APIC_BASE_MSR, determine the APIC mode and use the corresponding access method.

10.12.9 ICR Operation in x2APIC Mode

In x2APIC mode, the layout of the Interrupt Command Register is shown in Figure 10-12. The lower 32 bits of ICR in x2APIC mode is identical to the lower half of the ICR in xAPIC mode, except the Delivery Status bit is removed since it is not needed in x2APIC mode. The destination ID field is expanded to 32 bits in x2APIC mode.

To send an IPI using the ICR, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. Self IPIs can also be sent using the SELF IPI register (see Section 10.12.11).

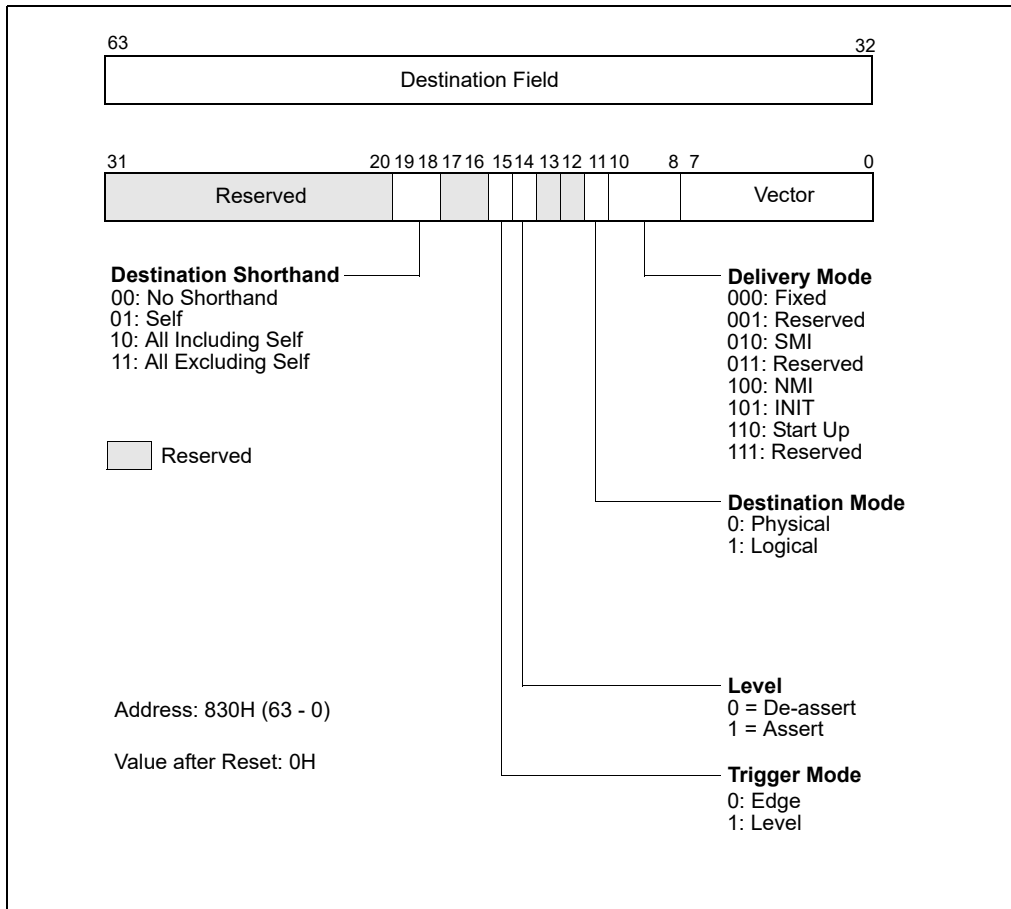


Figure 10-28. Interrupt Command Register (ICR) in x2APIC Mode

A single MSR write to the Interrupt Command Register is required for dispatching an interrupt in x2APIC mode. With the removal of the Delivery Status bit, system software no longer has a reason to read the ICR. It remains readable only to aid in debugging; however, software should not assume the value returned by reading the ICR is the last written value.

A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.

10.12.10 Determining IPI Destination in x2APIC Mode

10.12.10.1 Logical Destination Mode in x2APIC Mode

In x2APIC mode, the Logical Destination Register (LDR) is increased to 32 bits wide. It is a read-only register to system software. This 32-bit value is referred to as "logical x2APIC ID". System software accesses this register via the RDMSR instruction reading the MSR at address 80DH. Figure 10-29 provides the layout of the Logical Destination Register in x2APIC mode.

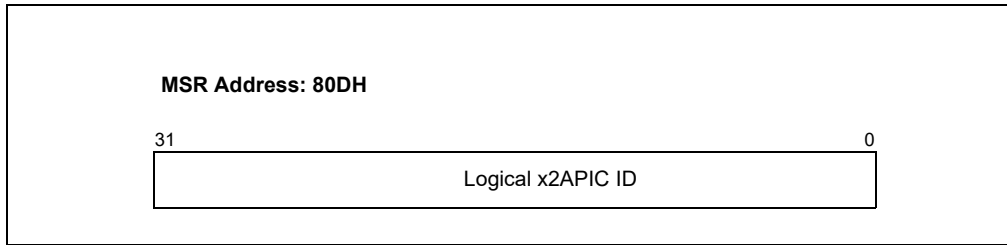


Figure 10-29. Logical Destination Register in x2APIC Mode

In the xAPIC mode, the Destination Format Register (DFR) through the MMIO interface determines the choice of a flat logical mode or a clustered logical mode. Flat logical mode is not supported in the x2APIC mode. Hence the Destination Format Register (DFR) is eliminated in x2APIC mode.

The 32-bit logical x2APIC ID field of LDR is partitioned into two sub-fields:

- Cluster ID (LDR[31:16]): is the address of the destination cluster
- Logical ID (LDR[15:0]): defines a logical ID of the individual local x2APIC within the cluster specified by LDR[31:16].

This layout enables $2^{16}-1$ clusters each with up to 16 unique logical IDs - effectively providing an addressability of $((2^{20}) - 16)$ processors in logical destination mode.

It is likely that processor implementations may choose to support less than 16 bits of the cluster ID or less than 16-bits of the Logical ID in the Logical Destination Register. However system software should be agnostic to the number of bits implemented in the cluster ID and logical ID sub-fields. The x2APIC hardware initialization will ensure that the appropriately initialized logical x2APIC IDs are available to system software and reads of non-implemented bits return zero. This is a read-only register that software must read to determine the logical x2APIC ID of the processor. Specifically, software can apply a 16-bit mask to the lowest 16 bits of the logical x2APIC ID to identify the logical address of a processor within a cluster without needing to know the number of implemented bits in cluster ID and Logical ID sub-fields. Similarly, software can create a message destination address for cluster model, by bit-Oring the Logical X2APIC ID (31:0) of processors that have matching Cluster ID(31:16).

To enable cluster ID assignment in a fashion that matches the system topology characteristics and to enable efficient routing of logical mode lowest priority device interrupts in link based platform interconnects, the LDR are initialized by hardware based on the value of x2APIC ID upon x2APIC state transitions. Details of this initialization are provided in Section 10.12.10.2.

10.12.10.2 Deriving Logical x2APIC ID from the Local x2APIC ID

In x2APIC mode, the 32-bit logical x2APIC ID, which can be read from LDR, is derived from the 32-bit local x2APIC ID. Specifically, the 16-bit logical ID sub-field is derived by shifting 1 by the lowest 4 bits of the x2APIC ID, i.e. Logical ID = $1 \ll x2APIC\ ID[3:0]$. The remaining bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

$$\text{Logical x2APIC ID} = [(x2APIC\ ID[19:4] \ll 16) | (1 \ll x2APIC\ ID[3:0])]$$

The use of the lowest 4 bits in the x2APIC ID implies that at least 16 APIC IDs are reserved for logical processors within a socket in multi-socket configurations. If more than 16 APIC IDs are reserved for logical processors in a socket/package then multiple cluster IDs can exist within the package.

The LDR initialization occurs whenever the x2APIC mode is enabled (see Section 10.12.5).

10.12.11 SELF IPI Register

SELF IPIs are used extensively by some system software. The x2APIC architecture introduces a new register interface. This new register is dedicated to the purpose of sending self-IPIs with the intent of enabling a highly optimized path for sending self-IPIs.

Figure 10-30 provides the layout of the SELF IPI register. System software only specifies the vector associated with the interrupt to be sent. The semantics of sending a self-IPI via the SELF IPI register are identical to sending a self targeted edge triggered fixed interrupt with the specified vector. Specifically the semantics are identical to the following settings for an inter-processor interrupt sent via the ICR - Destination Shorthand (ICR[19:18] = 01 (Self)), Trigger Mode (ICR[15] = 0 (Edge)), Delivery Mode (ICR[10:8] = 000 (Fixed)), Vector (ICR[7:0] = Vector).

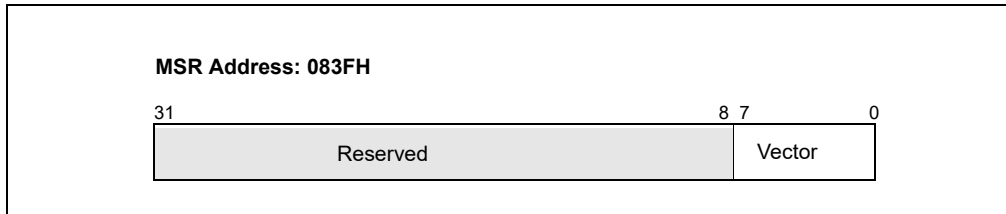


Figure 10-30. SELF IPI register

The SELF IPI register is a write-only register. A RDMSR instruction with address of the SELF IPI register causes a general-protection exception.

The handling and prioritization of a self-IPI sent via the SELF IPI register is architecturally identical to that for an IPI sent via the ICR from a legacy xAPIC unit. Specifically the state of the interrupt would be tracked via the Interrupt Request Register (IRR) and In Service Register (ISR) and Trigger Mode Register (TMR) as if it were received from the system bus. Also sending the IPI via the Self Interrupt Register ensures that interrupt is delivered to the processor core. Specifically completion of the WRMSR instruction to the SELF IPI register implies that the interrupt has been logged into the IRR. As expected for edge triggered interrupts, depending on the processor priority and readiness to accept interrupts, it is possible that interrupts sent via the SELF IPI register or via the ICR with identical vectors can be combined.

10.13 APIC BUS MESSAGE FORMATS

This section describes the message formats used when transmitting messages on the serial APIC bus. The information described here pertains only to the Pentium and P6 family processors.

10.13.1 Bus Message Formats

The local and I/O APICs transmit three types of messages on the serial APIC bus: EOI message, short message, and non-focused lowest priority message. The purpose of each type of message and its format are described below.

10.13.2 EOI Message

Local APICs send 14-cycle EOI messages to the I/O APIC to indicate that a level triggered interrupt has been accepted by the processor. This interrupt, in turn, is a result of software writing into the EOI register of the local APIC. Table 10-1 shows the cycles in an EOI message.

Table 10-1. EOI Message (14 Cycles)

Cycle	Bit1	Bit0	
1	1	1	11 = EOI
2	ArbID3	0	Arbitration ID bits 3 through 0

Table 10-1. EOI Message (14 Cycles) (Contd.)

Cycle	Bit1	Bit0	
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	V7	V6	Interrupt vector V7 - V0
7	V5	V4	
8	V3	V2	
9	V1	V0	
10	C	C	Checksum for cycles 6 - 9
11	0	0	
12	A	A	Status Cycle 0
13	A1	A1	Status Cycle 1
14	0	0	Idle

The checksum is computed for cycles 6 through 9. It is a cumulative sum of the 2-bit (Bit1:Bit0) logical data values. The carry out of all but the last addition is added to the sum. If any APIC computes a different checksum than the one appearing on the bus in cycle 10, it signals an error, driving 11 on the APIC bus during cycle 12. In this case, the APICs disregard the message. The sending APIC will receive an appropriate error indication (see Section 10.5.3, "Error Handling") and resend the message. The status cycles are defined in Table 10-4.

10.13.2.1 Short Message

Short messages (21-cycles) are used for sending fixed, NMI, SMI, INIT, start-up, ExtINT and lowest-priority-with-focus interrupts. Table 10-2 shows the cycles in a short message.

Table 10-2. Short Message (21 Cycles)

Cycle	Bit1	Bit0	
1	0	1	0 1 = normal
2	ArbID3	0	Arbitration ID bits 3 through 0
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	DM	M2	DM = Destination Mode
7	M1	M0	M2-M0 = Delivery mode
8	L	TM	L = Level, TM = Trigger Mode
9	V7	V6	V7-V0 = Interrupt Vector
10	V5	V4	
11	V3	V2	
12	V1	V0	
13	D7	D6	D7-D0 = Destination
14	D5	D4	
15	D3	D2	
16	D1	D0	

Table 10-2. Short Message (21 Cycles) (Contd.)

Cycle	Bit1	Bit0	
17	C	C	Checksum for cycles 6-16
18	0	0	
19	A	A	Status cycle 0
20	A1	A1	Status cycle 1
21	0	0	Idle

If the physical delivery mode is being used, then cycles 15 and 16 represent the APIC ID and cycles 13 and 14 are considered don't care by the receiver. If the logical delivery mode is being used, then cycles 13 through 16 are the 8-bit logical destination field.

For shorthands of "all-incl-self" and "all-excl-self," the physical delivery mode and an arbitration priority of 15 (D0:D3 = 1111) are used. The agent sending the message is the only one required to distinguish between the two cases. It does so using internal information.

When using lowest priority delivery with an existing focus processor, the focus processor identifies itself by driving 10 during cycle 19 and accepts the interrupt. This is an indication to other APICs to terminate arbitration. If the focus processor has not been found, the short message is extended on-the-fly to the non-focused lowest-priority message. Note that except for the EOI message, messages generating a checksum or an acceptance error (see Section 10.5.3, "Error Handling") terminate after cycle 21.

10.13.2.2 Non-focused Lowest Priority Message

These 34-cycle messages (see Table 10-3) are used in the lowest priority delivery mode when a focus processor is not present. Cycles 1 through 20 are same as for the short message. If during the status cycle (cycle 19) the state of the (A:A) flags is 10B, a focus processor has been identified, and the short message format is used (see Table 10-2). If the (A:A) flags are set to 00B, lowest priority arbitration is started and the 34-cycles of the non-focused lowest priority message are competed. For other combinations of status flags, refer to Section 10.13.2.3, "APIC Bus Status Cycles."

Table 10-3. Non-Focused Lowest Priority Message (34 Cycles)

Cycle	Bit0	Bit1	
1	0	1	0 1 = normal
2	ArbID3	0	Arbitration ID bits 3 through 0
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	DM	M2	DM = Destination mode
7	M1	M0	M2-M0 = Delivery mode
8	L	TM	L = Level, TM = Trigger Mode
9	V7	V6	V7-V0 = Interrupt Vector
10	V5	V4	
11	V3	V2	
12	V1	V0	
13	D7	D6	D7-D0 = Destination
14	D5	D4	
15	D3	D2	
16	D1	D0	

Table 10-3. Non-Focused Lowest Priority Message (34 Cycles) (Contd.)

Cycle	Bit0	Bit1	
17	C	C	Checksum for cycles 6-16
18	0	0	
19	A	A	Status cycle 0
20	A1	A1	Status cycle 1
21	P7	0	P7 - P0 = Inverted Processor Priority
22	P6	0	
23	P5	0	
24	P4	0	
25	P3	0	
26	P2	0	
27	P1	0	
28	P0	0	
29	ArbID3	0	Arbitration ID 3 -0
30	ArbID2	0	
31	ArbID1	0	
32	ArbID0	0	
33	A2	A2	Status Cycle
34	0	0	Idle

Cycles 21 through 28 are used to arbitrate for the lowest priority processor. The processors participating in the arbitration drive their inverted processor priority on the bus. Only the local APICs having free interrupt slots participate in the lowest priority arbitration. If no such APIC exists, the message will be rejected, requiring it to be tried at a later time.

Cycles 29 through 32 are also used for arbitration in case two or more processors have the same lowest priority. In the lowest priority delivery mode, all combinations of errors in cycle 33 (A2 A2) will set the “accept error” bit in the error status register (see Figure 10-9). Arbitration priority update is performed in cycle 20, and is not affected by errors detected in cycle 33. Only the local APIC that wins in the lowest priority arbitration, drives cycle 33. An error in cycle 33 will force the sender to resend the message.

10.13.2.3 APIC Bus Status Cycles

Certain cycles within an APIC bus message are status cycles. During these cycles the status flags (A:A) and (A1:A1) are examined. Table 10-4 shows how these status flags are interpreted, depending on the current delivery mode and existence of a focus processor.

Table 10-4. APIC Bus Status Cycles Interpretation

Delivery Mode	A Status	A1 Status	A2 Status	Update Arbid and Cycle#	Message Length	Retry
EOI	00: CS_OK	10: Accept	XX:	Yes, 13	14 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 13	14 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	14 Cycle	Yes
	11: CS_Error	XX:	XX:	No	14 Cycle	Yes
	10: Error	XX:	XX:	No	14 Cycle	Yes
	01: Error	XX:	XX:	No	14 Cycle	Yes
Fixed	00: CS_OK	10: Accept	XX:	Yes, 20	21 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 20	21 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	21 Cycle	Yes
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	10: Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes
NMI, SMI, INIT, ExtINT, Start-Up	00: CS_OK	10: Accept	XX:	Yes, 20	21 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 20	21 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	21 Cycle	Yes
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	10: Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes
Lowest	00: CS_OK, NoFocus	11: Do Lowest	10: Accept	Yes, 20	34 Cycle	No
	00: CS_OK, NoFocus	11: Do Lowest	11: Error	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	11: Do Lowest	0X: Error	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	10: End and Retry	XX:	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	0X: Error	XX:	No	34 Cycle	Yes
	10: CS_OK, Focus	XX:	XX:	Yes, 20	34 Cycle	No
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes

20. Updates to Chapter 11, Volume 3A

Change bars and green text show changes to Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

Changes to this chapter: Minor updates to various sections.

This chapter describes the memory cache and cache control mechanisms, the TLBs, and the store buffer in Intel 64 and IA-32 processors. It also describes the memory type range registers (MTRRs) introduced in the P6 family processors and how they are used to control caching of physical memory locations.

11.1 INTERNAL CACHES, TLBS, AND BUFFERS

The Intel 64 and IA-32 architectures support cache, translation look aside buffers (TLBs), and a store buffer for temporary on-chip (and external) storage of instructions and data. (Figure 11-1 shows the arrangement of caches, TLBs, and the store buffer for the Pentium 4 and Intel Xeon processors.) Table 11-1 shows the characteristics of these caches and buffers for the Pentium 4, Intel Xeon, P6 family, and Pentium processors. **The sizes and characteristics of these units are machine specific and may change in future versions of the processor.** The CPUID instruction returns the sizes and characteristics of the caches and buffers for the processor on which the instruction is executed. See “CPUID—CPU Identification” in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

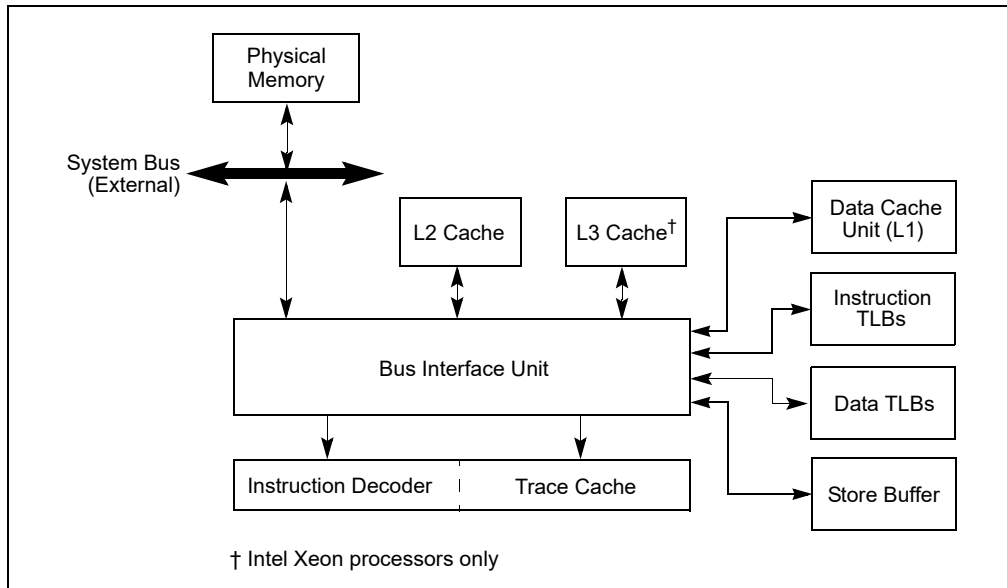


Figure 11-1. Cache Structure of the Pentium 4 and Intel Xeon Processors

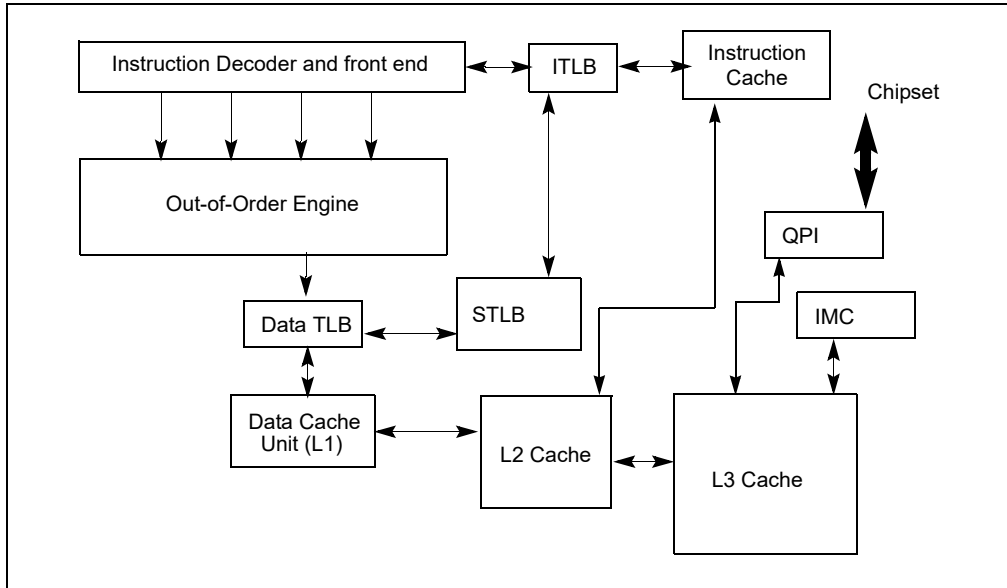


Figure 11-2. Cache Structure of the Intel Core i7 Processors

Figure 11-2 shows the cache arrangement of Intel Core i7 processor.

Table 11-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors

Cache or Buffer	Characteristics
Trace Cache ¹	<ul style="list-style-type: none"> ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst® microarchitecture): 12 Kμops, 8-way set associative. ▪ Intel Core i7, Intel Core 2 Duo, Intel® Atom™, Intel Core Duo, Intel Core Solo, Pentium M processor: not implemented. ▪ P6 family and Pentium processors: not implemented.
L1 Instruction Cache	<ul style="list-style-type: none"> ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): not implemented. ▪ Intel Core i7 processor: 32-KByte, 4-way set associative. ▪ Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M processor: 32-KByte, 8-way set associative. ▪ P6 family and Pentium processors: 8- or 16-KByte, 4-way set associative, 32-byte cache line size; 2-way set associative for earlier Pentium processors.
L1 Data Cache	<ul style="list-style-type: none"> ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 8-KByte, 4-way set associative, 64-byte cache line size. ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 16-KByte, 8-way set associative, 64-byte cache line size. ▪ Intel Atom processors: 24-KByte, 6-way set associative, 64-byte cache line size. ▪ Intel Core i7, Intel Core 2 Duo, Intel Core Duo, Intel Core Solo, Pentium M and Intel Xeon processors: 32-KByte, 8-way set associative, 64-byte cache line size. ▪ P6 family processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KBytes, 2-way set associative for earlier P6 family processors. ▪ Pentium processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KByte, 2-way set associative for earlier Pentium processors.

Table 11-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

Cache or Buffer	Characteristics
L2 Unified Cache	<ul style="list-style-type: none"> ▪ Intel Core 2 Duo and Intel Xeon processors: up to 4-MByte (or 4MBx2 in quadcore processors), 16-way set associative, 64-byte cache line size. ▪ Intel Core 2 Duo and Intel Xeon processors: up to 6-MByte (or 6MBx2 in quadcore processors), 24-way set associative, 64-byte cache line size. ▪ Intel Core i7, i5, i3 processors: 256KByte, 8-way set associative, 64-byte cache line size. ▪ Intel Atom processors: 512-KByte, 8-way set associative, 64-byte cache line size. ▪ Intel Core Duo, Intel Core Solo processors: 2-MByte, 8-way set associative, 64-byte cache line size ▪ Pentium 4 and Intel Xeon processors: 256, 512, 1024, or 2048-KByte, 8-way set associative, 64-byte cache line size, 128-byte sector size. ▪ Pentium M processor: 1 or 2-MByte, 8-way set associative, 64-byte cache line size. ▪ P6 family processors: 128-KByte, 256-KByte, 512-KByte, 1-MByte, or 2-MByte, 4-way set associative, 32-byte cache line size. ▪ Pentium processor (external optional): System specific, typically 256- or 512-KByte, 4-way set associative, 32-byte cache line size.
L3 Unified Cache	<ul style="list-style-type: none"> ▪ Intel Xeon processors: 512-KByte, 1-MByte, 2-MByte, or 4-MByte, 8-way set associative, 64-byte cache line size, 128-byte sector size. ▪ Intel Core i7 processor, Intel Xeon processor 5500: Up to 8MByte, 16-way set associative, 64-byte cache line size. ▪ Intel Xeon processor 5600: Up to 12MByte, 64-byte cache line size. ▪ Intel Xeon processor 7500: Up to 24MByte, 64-byte cache line size.
Instruction TLB (4-KByte Pages)	<ul style="list-style-type: none"> ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 128 entries, 4-way set associative. ▪ Intel Atom processors: 32-entries, fully associative. ▪ Intel Core i7, i5, i3 processors: 64-entries per thread (128-entries per core), 4-way set associative. ▪ Intel Core 2 Duo, Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative. ▪ P6 family processors: 32 entries, 4-way set associative. ▪ Pentium processor: 32 entries, 4-way set associative; fully set associative for Pentium processors with MMX technology.
Data TLB (4-KByte Pages)	<ul style="list-style-type: none"> ▪ Intel Core i7, i5, i3 processors, DTLB0: 64-entries, 4-way set associative. ▪ Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 256 entries, 4 ways. ▪ Intel Atom processors: 16-entry-per-thread micro-TLB, fully associative; 64-entry DTLB, 4-way set associative; 16-entry PDE cache, fully associative. ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 64 entry, fully set associative, shared with large page DTLB. ▪ Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative. ▪ Pentium and P6 family processors: 64 entries, 4-way set associative; fully set, associative for Pentium processors with MMX technology.
Instruction TLB (Large Pages)	<ul style="list-style-type: none"> ▪ Intel Core i7, i5, i3 processors: 7-entries per thread, fully associative. ▪ Intel Core 2 Duo processors: 4 entries, 4 ways. ▪ Pentium 4 and Intel Xeon processors: large pages are fragmented. ▪ Intel Core Duo, Intel Core Solo, Pentium M processor: 2 entries, fully associative. ▪ P6 family processors: 2 entries, fully associative. ▪ Pentium processor: Uses same TLB as used for 4-KByte pages.
Data TLB (Large Pages)	<ul style="list-style-type: none"> ▪ Intel Core i7, i5, i3 processors, DTLB0: 32-entries, 4-way set associative. ▪ Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 32 entries, 4 ways. ▪ Intel Atom processors: 8 entries, 4-way set associative. ▪ Pentium 4 and Intel Xeon processors: 64 entries, fully set associative; shared with small page data TLBs. ▪ Intel Core Duo, Intel Core Solo, Pentium M processor: 8 entries, fully associative. ▪ P6 family processors: 8 entries, 4-way set associative. ▪ Pentium processor: 8 entries, 4-way set associative; uses same TLB as used for 4-KByte pages in Pentium processors with MMX technology.
Second-level Unified TLB (4-KByte Pages)	<ul style="list-style-type: none"> ▪ Intel Core i7, i5, i3 processor, STLB: 512-entries, 4-way set associative.

Table 11-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

Cache or Buffer	Characteristics
Store Buffer	<ul style="list-style-type: none"> ▪ Intel Core i7, i5, i3 processors: 32 entries. ▪ Intel Core 2 Duo processors: 20 entries. ▪ Intel Atom processors: 8 entries, used for both WC and store buffers. ▪ Pentium 4 and Intel Xeon processors: 24 entries. ▪ Pentium M processor: 16 entries. ▪ P6 family processors: 12 entries. ▪ Pentium processor: 2 buffers, 1 entry each (Pentium processors with MMX technology have 4 buffers for 4 entries).
Write Combining (WC) Buffer	<ul style="list-style-type: none"> ▪ Intel Core 2 Duo processors: 8 entries. ▪ Intel Atom processors: 8 entries, used for both WC and store buffers. ▪ Pentium 4 and Intel Xeon processors: 6 or 8 entries. ▪ Intel Core Duo, Intel Core Solo, Pentium M processors: 6 entries. ▪ P6 family processors: 4 entries.

NOTES:

1 Introduced to the IA-32 architecture in the Pentium 4 and Intel Xeon processors.

Intel 64 and IA-32 processors may implement four types of caches: the trace cache, the level 1 (L1) cache, the level 2 (L2) cache, and the level 3 (L3) cache. See Figure 11-1. Cache availability is described below:

- **Intel Core i7, i5, i3 processor Family and Intel Xeon processor Family based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache. Each processor core has its own L1 and L2. The L3 cache is an inclusive, unified data and instruction cache, shared by all processor cores inside a physical package. No trace cache is implemented.
- **Intel® Core™ 2 processor family and Intel® Xeon® processor family based on Intel® Core™ microarchitecture** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip; it is shared between two processor cores in a dual-core processor implementation. Quad-core processors have two L2, each shared by two processor cores. No trace cache is implemented.
- **Intel® Atom™ processor** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache is located on the processor chip. No trace cache is implemented.
- **Intel® Core™ Solo and Intel® Core™ Duo processors** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip. It is shared between two processor cores in a dual-core processor implementation. No trace cache is implemented.
- **Pentium® 4 and Intel® Xeon® processors Based on Intel NetBurst® microarchitecture** — The trace cache caches decoded instructions (μ ops) from the instruction decoder and the L1 cache contains data. The L2 and L3 caches are unified data and instruction caches located on the processor chip. Dualcore processors have two L2, one in each processor core. Note that the L3 cache is only implemented on some Intel Xeon processors.
- **P6 family processors** — The L1 cache is divided into two sections: one dedicated to caching instructions (pre-decoded instructions) and the other to caching data. The L2 cache is a unified data and instruction cache located on the processor chip. P6 family processors do not implement a trace cache.
- **Pentium® processors** — The L1 cache has the same structure as on P6 family processors. There is no trace cache. The L2 cache is a unified data and instruction cache external to the processor chip on earlier Pentium processors and implemented on the processor chip in later Pentium processors. For Pentium processors where the L2 cache is external to the processor, access to the cache is through the system bus.

For Intel Core i7 processors and processors based on Intel Core, Intel Atom, and Intel NetBurst microarchitectures, Intel Core Duo, Intel Core Solo and Pentium M processors, the cache lines for the L1 and L2 caches (and L3 caches if supported) are 64 bytes wide. The processor always reads a cache line from system memory beginning on a 64-byte boundary. (A 64-byte aligned cache line begins at an address with its 6 least-significant bits clear.) A cache

line can be filled from memory with a 8-transfer burst transaction. The caches do not support partially-filled cache lines, so caching even a single doubleword requires caching an entire line.

The L1 and L2 cache lines in the P6 family and Pentium processors are 32 bytes wide, with cache line reads from system memory beginning on a 32-byte boundary (5 least-significant bits of a memory address clear.) A cache line can be filled from memory with a 4-transfer burst transaction. Partially-filled cache lines are not supported.

The trace cache in processors based on Intel NetBurst microarchitecture is available in all execution modes: protected mode, system management mode (SMM), and real-address mode. The L1,L2, and L3 caches are also available in all execution modes; however, use of them must be handled carefully in SMM (see Section 34.4.2, "SMRAM Caching").

The TLBs store the most recently used page-directory and page-table entries. They speed up memory accesses when paging is enabled by reducing the number of memory accesses that are required to read the page tables stored in system memory. The TLBs are divided into four groups: instruction TLBs for 4-KByte pages, data TLBs for 4-KByte pages; instruction TLBs for large pages (2-MByte, 4-MByte or 1-GByte pages), and data TLBs for large pages. The TLBs are normally active only in protected mode with paging enabled. When paging is disabled or the processor is in real-address mode, the TLBs maintain their contents until explicitly or implicitly flushed (see Section 11.9, "Invalidating the Translation Lookaside Buffers (TLBs)").

Processors based on Intel Core microarchitectures implement one level of instruction TLB and two levels of data TLB. Intel Core i7 processor provides a second-level unified TLB.

The store buffer is associated with the processors instruction execution units. It allows writes to system memory and/or the internal caches to be saved and in some cases combined to optimize the processor's bus accesses. The store buffer is always enabled in all execution modes.

The processor's caches are for the most part transparent to software. When enabled, instructions and data flow through these caches without the need for explicit software control. However, knowledge of the behavior of these caches may be useful in optimizing software performance. For example, knowledge of cache dimensions and replacement algorithms gives an indication of how large of a data structure can be operated on at once without causing cache thrashing.

In multiprocessor systems, maintenance of cache consistency may, in rare circumstances, require intervention by system software. For these rare cases, the processor provides privileged cache control instructions for use in flushing caches and forcing memory ordering.

There are several instructions that software can use to improve the performance of the L1, L2, and L3 caches, including the PREFETCHh, CLFLUSH, and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD). The use of these instructions are discussed in Section 11.5.5, "Cache Management Instructions."

11.2 CACHING TERMINOLOGY

IA-32 processors (beginning with the Pentium processor) and Intel 64 processors use the MESI (modified, exclusive, shared, invalid) cache protocol to maintain consistency with internal caches and caches in other processors (see Section 11.4, "Cache Control Protocol").

When the processor recognizes that an operand being read from memory is cacheable, the processor reads an entire cache line into the appropriate cache (L1, L2, L3, or all). This operation is called a **cache line fill**. If the memory location containing that operand is still cached the next time the processor attempts to access the operand, the processor can read the operand from the cache instead of going back to memory. This operation is called a **cache hit**.

When the processor attempts to write an operand to a cacheable area of memory, it first checks if a cache line for that memory location exists in the cache. If a valid cache line does exist, the processor (depending on the write policy currently in force) can write the operand into the cache instead of writing it out to system memory. This operation is called a **write hit**. If a write misses the cache (that is, a valid cache line is not present for area of memory being written to), the processor performs a cache line fill, write allocation. Then it writes the operand into the cache line and (depending on the write policy currently in force) can also write it out to memory. If the operand is to be written out to memory, it is written first into the store buffer, and then written from the store buffer to memory when the system bus is available. (Note that for the Pentium processor, write misses do not result in a cache line fill; they always result in a write to memory. For this processor, only read misses result in cache line fills.)

When operating in an MP system, IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors have the ability to **snoop** other processor's accesses to system memory and to their internal caches. They use this snooping ability to keep their internal caches consistent both with system memory and with the caches in other processors on the bus. For example, in the Pentium and P6 family processors, if through snooping one processor detects that another processor intends to write to a memory location that it currently has cached in **shared state**, the snooping processor will invalidate its cache line forcing it to perform a cache line fill the next time it accesses the same memory location.

Beginning with the P6 family processors, if a processor detects (through snooping) that another processor is trying to access a memory location that it has modified in its cache, but has not yet written back to system memory, the snooping processor will signal the other processor (by means of the HITM# signal) that the cache line is held in modified state and will perform an implicit write-back of the modified data. The implicit write-back is transferred directly to the initial requesting processor and snooped by the memory controller to assure that system memory has been updated. Here, the processor with the valid data may pass the data to the other processors without actually writing it to system memory; however, it is the responsibility of the memory controller to snoop this operation and update memory.

11.3 METHODS OF CACHING AVAILABLE

The processor allows any area of system memory to be cached in the L1, L2, and L3 caches. In individual pages or regions of system memory, it allows the type of caching (also called **memory type**) to be specified (see Section 11.5). Memory types currently defined for the Intel 64 and IA-32 architectures are (see Table 11-2):

- **Strong Uncacheable (UC)** —System memory locations are not cached. All reads and writes appear on the system bus and are executed in program order without reordering. No speculative memory accesses, page-table walks, or prefetches of speculated branch targets are made. This type of cache-control is useful for memory-mapped I/O devices. When used with normal RAM, it greatly reduces processor performance.

NOTE

The behavior of **x87 and SIMD instructions referencing memory** is implementation dependent. In some implementations, accesses to UC memory may occur more than once. To ensure predictable behavior, use loads and stores of general purpose registers to access UC memory that may have read or write side effects.

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering
Uncacheable (UC-)	No	No	No	Strong Ordering. Can only be selected through the PAT. Can be overridden by WC in MTRRs.
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.
Write Protected (WP)	Yes for reads; no for writes	No	Yes	Speculative Processor Ordering. Available by programming MTRRs.

- **Uncacheable (UC-)** — Has same characteristics as the strong uncacheable (UC) memory type, except that this memory type can be overridden by programming the MTRRs for the WC memory type. This memory type is available in processor families starting from the Pentium III processors and can only be selected through the PAT.

- **Write Combining (WC)** — System memory locations are not cached (as with uncacheable memory) and coherency is not enforced by the processor's bus coherency protocol. Speculative reads are allowed. Writes may be delayed and combined in the write combining buffer (WC buffer) to reduce memory accesses. If the WC buffer is partially filled, the writes may be delayed until the next occurrence of a serializing event; such as an SFENCE or MFENCE instruction, CPUID or other serializing instruction, a read or write to uncached memory, an interrupt occurrence, or an execution of a LOCK instruction (including one with an XACQUIRE or XRELEASE prefix). In addition, an execution of the XEND instruction (to end a transactional region) evicts any writes that were buffered before the corresponding execution of the XBEGIN instruction (to begin the transactional region) before evicting any writes that were performed inside the transactional region.

This type of cache-control is appropriate for video frame buffers, where the order of writes is unimportant as long as the writes update memory so they can be seen on the graphics display. See Section 11.3.1, "Buffering of Write Combining Memory Locations," for more information about caching the WC memory type. This memory type is available in the Pentium Pro and Pentium II processors by programming the MTRRs; or in processor families starting from the Pentium III processors by programming the MTRRs or by selecting it through the PAT.

- **Write-through (WT)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and valid cache lines are either filled or invalidated. Write combining is allowed. This type of cache-control is appropriate for frame buffers or when there are devices on the system bus that access system memory, but do not perform snooping of memory accesses. It enforces coherency between caches in the processors and system memory.
- **Write-back (WB)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. Write misses cause cache line fills (in processor families starting with the P6 family processors), and writes are performed entirely in the cache, when possible. Write combining is allowed. The write-back memory type reduces bus traffic by eliminating many unnecessary writes to system memory. Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later, when a write-back operation is performed. Write-back operations are triggered when cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. They also are triggered by the mechanisms used to maintain cache consistency. This type of cache-control provides the best performance, but it requires that all devices that access system memory on the system bus be able to snoop memory accesses to insure system memory and cache coherency.
- **Write protected (WP)** — Reads come from cache lines when possible, and read misses cause cache fills. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated. Speculative reads are allowed. This memory type is available in processor families starting from the P6 family processors by programming the MTRRs (see Table 11-6).

Table 11-3 shows which of these caching methods are available in the Pentium, P6 Family, Pentium 4, and Intel Xeon processors.

Table 11-3. Methods of Caching Available in Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 Family, and Pentium Processors

Memory Type	Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon Processors	P6 Family Processors	Pentium Processor
Strong Uncacheable (UC)	Yes	Yes	Yes
Uncacheable (UC-)	Yes	Yes*	No
Write Combining (WC)	Yes	Yes	No
Write Through (WT)	Yes	Yes	Yes
Write Back (WB)	Yes	Yes	Yes
Write Protected (WP)	Yes	Yes	No

NOTE:

* Introduced in the Pentium III processor; not available in the Pentium Pro or Pentium II processors

11.3.1 Buffering of Write Combining Memory Locations

Writes to the WC memory type are not cached in the typical sense of the word cached. They are retained in an internal write combining buffer (WC buffer) that is separate from the internal L1, L2, and L3 caches and the store buffer. The WC buffer is not snooped and thus does not provide data coherency. Buffering of writes to WC memory is done to allow software a small window of time to supply more modified data to the WC buffer while remaining as non-intrusive to software as possible. The buffering of writes to WC memory also causes data to be collapsed; that is, multiple writes to the same memory location will leave the last data written in the location and the other writes will be lost.

The size and structure of the WC buffer is not architecturally defined. For the Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon processors; the WC buffer is made up of several 64-byte WC buffers. For the P6 family processors, the WC buffer is made up of several 32-byte WC buffers.

When software begins writing to WC memory, the processor begins filling the WC buffers one at a time. When one or more WC buffers has been filled, the processor has the option of evicting the buffers to system memory. The protocol for evicting the WC buffers is implementation dependent and should not be relied on by software for system memory coherency. When using the WC memory type, software **must** be sensitive to the fact that the writing of data to system memory is being delayed and **must** deliberately empty the WC buffers when system memory coherency is required.

Once the processor has started to evict data from the WC buffer into system memory, it will make a bus-transaction style decision based on how much of the buffer contains valid data. If the buffer is full (for example, all bytes are valid), the processor will execute a burst-write transaction on the bus. This results in all 32 bytes (P6 family processors) or 64 bytes (Pentium 4 and more recent processor) being transmitted on the data bus in a single burst transaction. If one or more of the WC buffer's bytes are invalid (for example, have not been written by software), the processor will transmit the data to memory using "partial write" transactions (one chunk at a time, where a "chunk" is 8 bytes).

This will result in a maximum of 4 partial write transactions (for P6 family processors) or 8 partial write transactions (for the Pentium 4 and more recent processors) for one WC buffer of data sent to memory.

The WC memory type is weakly ordered by definition. Once the eviction of a WC buffer has started, the data is subject to the weak ordering semantics of its definition. Ordering is not maintained between the successive allocation/deallocation of WC buffers (for example, writes to WC buffer 1 followed by writes to WC buffer 2 may appear as buffer 2 followed by buffer 1 on the system bus). When a WC buffer is evicted to memory as partial writes there is no guaranteed ordering between successive partial writes (for example, a partial write for chunk 2 may appear on the bus before the partial write for chunk 1 or vice versa).

The only elements of WC propagation to the system bus that are guaranteed are those provided by transaction atomicity. For example, with a P6 family processor, a completely full WC buffer will always be propagated as a single 32-bit burst transaction using any chunk order. In a WC buffer eviction where data will be evicted as partials, all data contained in the same chunk (0 mod 8 aligned) will be propagated simultaneously. Likewise, for more recent processors starting with those based on Intel NetBurst microarchitectures, a full WC buffer will always be propagated as a single burst transactions, using any chunk order within a transaction. For partial buffer propagations, all data contained in the same chunk will be propagated simultaneously.

11.3.2 Choosing a Memory Type

The simplest system memory model does not use memory-mapped I/O with read or write side effects, does not include a frame buffer, and uses the write-back memory type for all memory. An I/O agent can perform direct memory access (DMA) to write-back memory and the cache protocol maintains cache coherency.

A system can use strong uncacheable memory for other memory-mapped I/O, and should always use strong uncacheable memory for memory-mapped I/O with read side effects.

Dual-ported memory can be considered a write side effect, making relatively prompt writes desirable, because those writes cannot be observed at the other port until they reach the memory agent. A system can use strong uncacheable, uncacheable, write-through, or write-combining memory for frame buffers or dual-ported memory that contains pixel values displayed on a screen. Frame buffer memory is typically large (a few megabytes) and is usually written more than it is read by the processor. Using strong uncacheable memory for a frame buffer generates very large amounts of bus traffic, because operations on the entire buffer are implemented using partial writes rather than line writes. Using write-through memory for a frame buffer can displace almost all other useful cached

lines in the processor's L2 and L3 caches and L1 data cache. Therefore, systems should use write-combining memory for frame buffers whenever possible.

Software can use page-level cache control, to assign appropriate effective memory types when software will not access data structures in ways that benefit from write-back caching. For example, software may read a large data structure once and not access the structure again until the structure is rewritten by another agent. Such a large data structure should be marked as uncacheable, or reading it will evict cached lines that the processor will be referencing again.

A similar example would be a write-only data structure that is written to (to export the data to another agent), but never read by software. Such a structure can be marked as uncacheable, because software never reads the values that it writes (though as uncacheable memory, it will be written using partial writes, while as write-back memory, it will be written using line writes, which may not occur until the other agent reads the structure and triggers implicit write-backs).

On the Pentium III, Pentium 4, and more recent processors, new instructions are provided that give software greater control over the caching, prefetching, and the write-back characteristics of data. These instructions allow software to use weakly ordered or processor ordered memory types to improve processor performance, but when necessary to force strong ordering on memory reads and/or writes. They also allow software greater control over the caching of data. For a description of these instructions and their intended use, see Section 11.5.5, "Cache Management Instructions."

11.3.3 Code Fetches in Uncacheable Memory

Programs may execute code from uncacheable (UC) memory, but the implications are different from accessing data in UC memory. When doing code fetches, the processor never transitions from cacheable code to UC code speculatively. It also never speculatively fetches branch targets that result in UC code.

The processor may fetch the same UC cache line multiple times in order to decode an instruction once. It may decode consecutive UC instructions in a cacheline without fetching between each instruction. It may also fetch additional cachelines from the same or a consecutive 4-KByte page in order to decode one non-speculative UC instruction (this can be true even when the instruction is contained fully in one line).

Because of the above and because cacheline sizes may change in future processors, software should avoid placing memory-mapped I/O with read side effects in the same page or in a subsequent page used to execute UC code.

11.4 CACHE CONTROL PROTOCOL

The following section describes the cache control protocol currently defined for the Intel 64 and IA-32 architectures.

In the L1 data cache and in the L2/L3 unified caches, the MESI (modified, exclusive, shared, invalid) cache protocol maintains consistency with caches of other processors. The L1 data cache and the L2/L3 unified caches have two MESI status flags per cache line. Each line can be marked as being in one of the states defined in Table 11-4. In general, the operation of the MESI protocol is transparent to programs.

Table 11-4. MESI Cache Line States

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	Out of date	Valid	Valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	Does not go to the system bus.	Does not go to the system bus.	Causes the processor to gain exclusive ownership of the line.	Goes directly to the system bus.

The L1 instruction cache in P6 family processors implements only the “SI” part of the MESI protocol, because the instruction cache is not writable. The instruction cache monitors changes in the data cache to maintain consistency between the caches when instructions are modified. See Section 11.6, “Self-Modifying Code,” for more information on the implications of caching instructions.

11.5 CACHE CONTROL

The Intel 64 and IA-32 architectures provide a variety of mechanisms for controlling the caching of data and instructions and for controlling the ordering of reads and writes between the processor, the caches, and memory. These mechanisms can be divided into two groups:

- **Cache control registers and bits** — The Intel 64 and IA-32 architectures define several dedicated registers and various bits within control registers and page- and directory-table entries that control the caching system memory locations in the L1, L2, and L3 caches. These mechanisms control the caching of virtual memory pages and of regions of physical memory.
- **Cache control and memory ordering instructions** — The Intel 64 and IA-32 architectures provide several instructions that control the caching of data, the ordering of memory reads and writes, and the prefetching of data. These instructions allow software to control the caching of specific data structures, to control memory coherency for specific locations in memory, and to force strong memory ordering at specific locations in a program.

The following sections describe these two groups of cache control mechanisms.

11.5.1 Cache Control Registers and Bits

Figure 11-3 depicts cache-control mechanisms in IA-32 processors. Other than for the matter of memory address space, these work the same in Intel 64 processors.

The Intel 64 and IA-32 architectures provide the following cache-control registers and bits for use in enabling or restricting caching to various pages or regions in memory:

- **CD flag, bit 30 of control register CR0** — Controls caching of system memory locations (see Section 2.5, “Control Registers”). If the CD flag is clear, caching is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. When the CD flag is set, caching is restricted in the processor’s caches (cache hierarchy) for the P6 and more recent processor families and prevented for the Pentium processor (see note below). With the CD flag set, however, the caches will still respond to snoop traffic. Caches should be explicitly flushed to insure memory coherency. For highest processor performance, both the CD and the NW flags in control register CR0 should be cleared. Table 11-5 shows the interaction of the CD and NW flags.

The effect of setting the CD flag is somewhat different for processor families starting with P6 family than the Pentium processor (see Table 11-5). To insure memory coherency after the CD flag is set, the caches should be explicitly flushed (see Section 11.5.3, “Preventing Caching”). Setting the CD flag for the P6 and more recent processor families modify cache line fill and update behaviour. Also, setting the CD flag on these processors do not force strict ordering of memory accesses unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 8.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

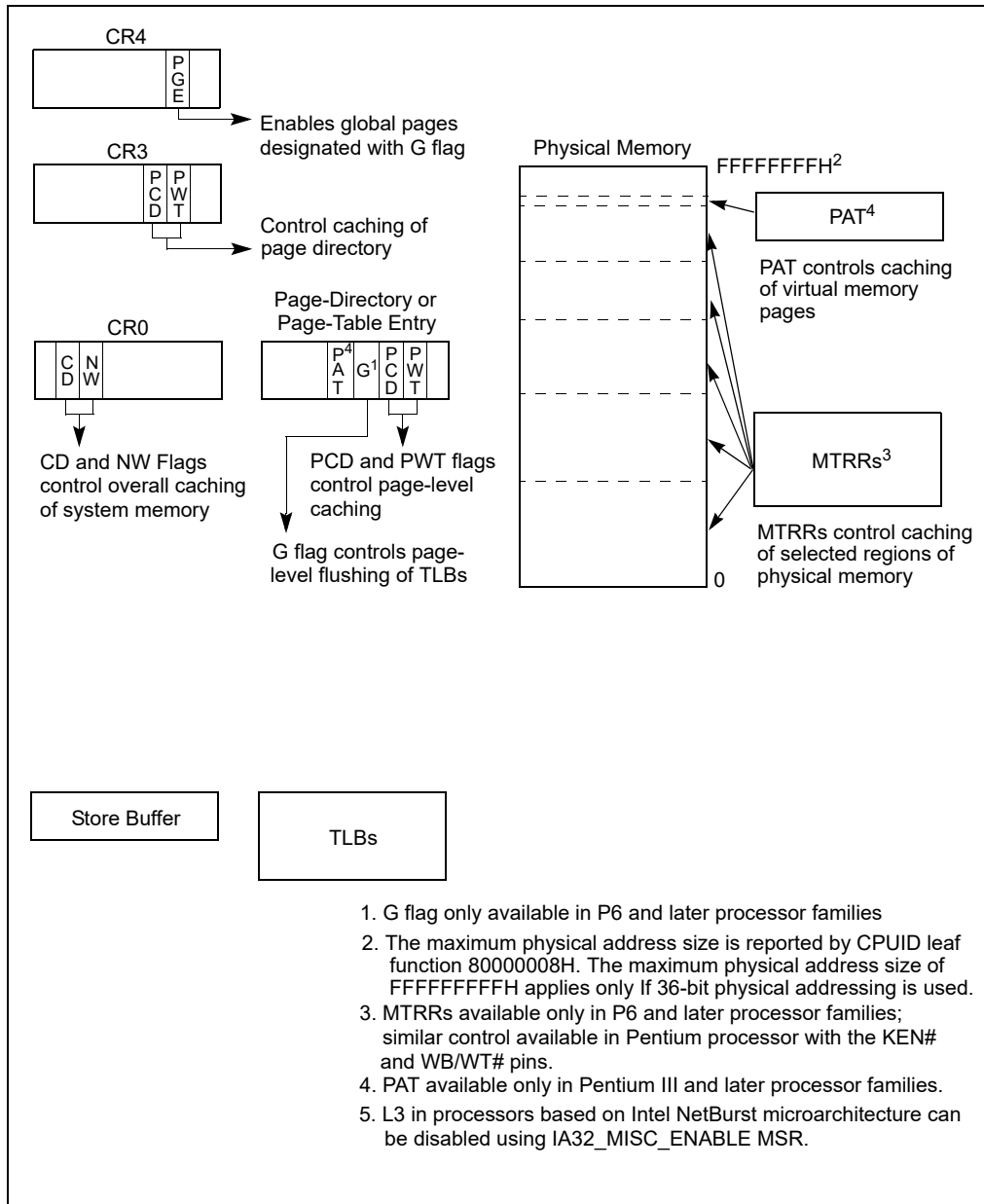


Figure 11-3. Cache-Control Registers and Bits Available in Intel 64 and IA-32 Processors

Table 11-5. Cache Operating Modes

CD	NW	Caching and Read/Write Policy	L1	L2/L3 ¹
0	0	<p>Normal Cache Mode. Highest performance cache operation.</p> <ul style="list-style-type: none"> ▪ Read hits access the cache; read misses may cause replacement. ▪ Write hits update the cache. ▪ Only writes to shared lines and write misses update system memory. ▪ Write misses cause cache line fills. ▪ Write hits can change shared lines to modified under control of the MTRRs and with associated read invalidation cycle. ▪ (Pentium processor only.) Write misses do not cause cache line fills. ▪ (Pentium processor only.) Write hits can change shared lines to exclusive under control of WB/WT#. ▪ Invalidation is allowed. ▪ External snoop traffic is supported. 	<p>Yes Yes Yes Yes Yes Yes Yes Yes</p>	<p>Yes Yes Yes Yes</p>
0	1	<p>Invalid setting. Generates a general-protection exception (#GP) with an error code of 0.</p>	NA	NA
1	0	<p>No-fill Cache Mode. Memory coherency is maintained.³</p> <ul style="list-style-type: none"> ▪ (Pentium 4 and later processor families.) State of processor after a power up or reset. ▪ Read hits access the cache; read misses do not cause replacement (see Pentium 4 and Intel Xeon processors reference below). ▪ Write hits update the cache. ▪ Only writes to shared lines and write misses update system memory. ▪ Write misses access memory. ▪ Write hits can change shared lines to exclusive under control of the MTRRs and with associated read invalidation cycle. ▪ (Pentium processor only.) Write hits can change shared lines to exclusive under control of the WB/WT#. ▪ (P6 and later processor families only.) Strict memory ordering is not enforced unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 7.2.4., "Strengthening or Weakening the Memory Ordering Model"). ▪ Invalidation is allowed. ▪ External snoop traffic is supported. 	<p>Yes Yes Yes Yes Yes Yes Yes Yes</p>	<p>Yes Yes Yes Yes Yes Yes</p>
1	1	<p>Memory coherency is not maintained.^{2, 3}</p> <ul style="list-style-type: none"> ▪ (P6 family and Pentium processors.) State of the processor after a power up or reset. ▪ Read hits access the cache; read misses do not cause replacement. ▪ Write hits update the cache and change exclusive lines to modified. ▪ Shared lines remain shared after write hit. ▪ Write misses access memory. ▪ Invalidation is inhibited when snooping; but is allowed with INVD and WBINVD instructions. ▪ External snoop traffic is supported. 	<p>Yes Yes Yes Yes Yes Yes No</p>	<p>Yes Yes Yes Yes Yes Yes</p>

NOTES:

1. The L2/L3 column in this table is definitive for the Pentium 4, Intel Xeon, and P6 family processors. It is intended to represent what could be implemented in a system based on a Pentium processor with an external, platform specific, write-back L2 cache.
2. The Pentium 4 and more recent processor families do not support this mode; setting the CD and NW bits to 1 selects the no-fill cache mode.
3. Not supported In Intel Atom processors. If CD = 1 in an Intel Atom processor, caching is disabled.

- **NW flag, bit 29 of control register CR0** — Controls the write policy for system memory locations (see Section 2.5, “Control Registers”). If the NW and CD flags are clear, write-back is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. Table 11-5 shows how the other combinations of CD and NW flags affects caching.

NOTES

For the Pentium 4 and Intel Xeon processors, the NW flag is a don't care flag; that is, when the CD flag is set, the processor uses the no-fill cache mode, regardless of the setting of the NW flag.

For Intel Atom processors, the NW flag is a don't care flag; that is, when the CD flag is set, the processor disables caching, regardless of the setting of the NW flag.

For the Pentium processor, when the L1 cache is disabled (the CD and NW flags in control register CR0 are set), external snoops are accepted in DP (dual-processor) systems and inhibited in uniprocessor systems.

When snoops are inhibited, address parity is not checked and APCHK# is not asserted for a corrupt address; however, when snoops are accepted, address parity is checked and APCHK# is asserted for corrupt addresses.

- **PCD and PWT flags in paging-structure entries** — Control the memory type used to access paging structures and pages (see Section 4.9, “Paging and Memory Typing”).
- **PCD and PWT flags in control register CR3** — Control the memory type used to access the first paging structure of the current paging-structure hierarchy (see Section 4.9, “Paging and Memory Typing”).
- **G (global) flag in the page-directory and page-table entries (introduced to the IA-32 architecture in the P6 family processors)** — Controls the flushing of TLB entries for individual pages. See Section 4.10, “Caching Translation Information,” for more information about this flag.
- **PGE (page global enable) flag in control register CR4** — Enables the establishment of global pages with the G flag. See Section 4.10, “Caching Translation Information,” for more information about this flag.
- **Memory type range registers (MTRRs) (introduced in P6 family processors)** — Control the type of caching used in specific regions of physical memory. Any of the caching types described in Section 11.3, “Methods of Caching Available,” can be selected. See Section 11.11, “Memory Type Range Registers (MTRRs),” for a detailed description of the MTRRs.
- **Page Attribute Table (PAT) MSR (introduced in the Pentium III processor)** — Extends the memory typing capabilities of the processor to permit memory types to be assigned on a page-by-page basis (see Section 11.12, “Page Attribute Table (PAT)”).
- **Third-Level Cache Disable flag, bit 6 of the IA32_MISC_ENABLE MSR (Available only in processors based on Intel NetBurst microarchitecture)** — Allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches.
- **KEN# and WB/WT# pins (Pentium processor)** — Allow external hardware to control the caching method used for specific areas of memory. They perform similar (but not identical) functions to the MTRRs in the P6 family processors.
- **PCD and PWT pins (Pentium processor)** — These pins (which are associated with the PCD and PWT flags in control register CR3 and in the page-directory and page-table entries) permit caching in an external L2 cache to be controlled on a page-by-page basis, consistent with the control exercised on the L1 cache of these processors. The P6 and more recent processor families do not provide these pins because the L2 cache is internal to the chip package.

11.5.2 Precedence of Cache Controls

The cache control flags and MTRRs operate hierarchically for restricting caching. That is, if the CD flag is set, caching is prevented globally (see Table 11-5). If the CD flag is clear, the page-level cache control flags and/or the MTRRs can be used to restrict caching. If there is an overlap of page-level and MTRR caching controls, the mechanism that prevents caching has precedence. For example, if an MTRR makes a region of system memory uncacheable, a page-level caching control cannot be used to enable caching for a page in that region. The converse is also

true; that is, if a page-level caching control designates a page as uncacheable, an MTRR cannot be used to make the page cacheable.

In cases where there is an overlap in the assignment of the write-back and write-through caching policies to a page and a region of memory, the write-through policy takes precedence. The write-combining policy (which can only be assigned through an MTRR or the PAT) takes precedence over either write-through or write-back.

The selection of memory types at the page level varies depending on whether PAT is being used to select memory types for pages, as described in the following sections.

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. Using IA32_MISC_ENABLE[bit 6] takes precedence over the CD flag, MTRRs, and PAT for the L3 cache in those processors. That is, when the third-level cache disable flag is set (cache disabled), the other cache controls have no effect on the L3 cache; when the flag is clear (enabled), the cache controls have the same effect on the L3 cache as they have on the L1 and L2 caches.

IA32_MISC_ENABLE[bit 6] is not supported in Intel Core i7 processors, nor processors based on Intel Core, and Intel Atom microarchitectures.

11.5.2.1 Selecting Memory Types for Pentium Pro and Pentium II Processors

The Pentium Pro and Pentium II processors do not support the PAT. Here, the effective memory type for a page is selected with the MTRRs and the PCD and PWT bits in the page-table or page-directory entry for the page. Table 11-6 describes the mapping of MTRR memory types and page-level caching attributes to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear). Combinations that appear in gray are implementation-defined for the Pentium Pro and Pentium II processors. System designers are encouraged to avoid these implementation-defined combinations.

Table 11-6. Effective Page-Level Memory Type for Pentium Pro and Pentium II Processors

MTRR Memory Type ¹	PCD Value	PWT Value	Effective Memory Type
UC	X	X	UC
WC	0	0	WC
	0	1	WC
	1	0	WC
	1	1	UC
WT	0	X	WT
	1	X	UC
WP	0	0	WP
	0	1	WP
	1	0	WC
	1	1	UC
WB	0	0	WB
	0	1	WT
	1	X	UC

NOTE:

1. These effective memory types also apply to the Pentium 4, Intel Xeon, and Pentium III processors when the PAT bit is not used (set to 0) in page-table and page-directory entries.

When normal caching is in effect, the effective memory type shown in Table 11-6 is determined using the following rules:

1. If the PCD and PWT attributes for the page are both 0, then the effective memory type is identical to the MTRR-defined memory type.

2. If the PCD flag is set, then the effective memory type is UC.
3. If the PCD flag is clear and the PWT flag is set, the effective memory type is WT for the WB memory type and the MTRR-defined memory type for all other memory types.
4. Setting the PCD and PWT flags to opposite values is considered model-specific for the WP and WC memory types and architecturally-defined for the WB, WT, and UC memory types.

11.5.2.2 Selecting Memory Types for Pentium III and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M, Pentium 4, Intel Xeon, and Pentium III processors use the PAT to select effective page-level memory types. Here, a memory type for a page is selected by the MTRRs and the value in a PAT entry that is selected with the PAT, PCD and PWT bits in a page-table or page-directory entry (see Section 11.12.3, “Selecting a Memory Type from the PAT”). Table 11-7 describes the mapping of MTRR memory types and PAT entry types to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear).

Table 11-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families

MTRR Memory Type	PAT Entry Value	Effective Memory Type
UC	UC	UC ¹
	UC-	UC ¹
	WC	WC
	WT	UC ¹
	WB	UC ¹
	WP	UC ¹
WC	UC	UC ²
	UC-	WC
	WC	WC
	WT	UC ^{2,3}
	WB	WC
	WP	UC ^{2,3}
WT	UC	UC ²
	UC-	UC ²
	WC	WC
	WT	WT
	WB	WT
	WP	WP ³
WB	UC	UC ²
	UC-	UC ²
	WC	WC
	WT	WT
	WB	WB
	WP	WP

Table 11-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families (Contd.)

MTRR Memory Type	PAT Entry Value	Effective Memory Type
WP	UC	UC ²
	UC-	WC ³
	WC	WC
	WT	WT ³
	WB	WP
	WP	WP

NOTES:

1. The UC attribute comes from the MTRRs and the processors are not required to snoop their caches since the data could never have been cached. This attribute is preferred for performance reasons.
2. The UC attribute came from the page-table or page-directory entry and processors are required to check their caches because the data may be cached due to page aliasing, which is not recommended.
3. These combinations were specified as “undefined” in previous editions of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. However, all processors that support both the PAT and the MTRRs determine the effective page-level memory types for these combinations as given.

11.5.2.3 Writing Values Across Pages with Different Memory Types

If two adjoining pages in memory have different memory types, and a word or longer operand is written to a memory location that crosses the page boundary between those two pages, the operand might be written to memory twice. This action does not present a problem for writes to actual memory; however, if a device is mapped the memory space assigned to the pages, the device might malfunction.

11.5.3 Preventing Caching

To disable the L1, L2, and L3 caches after they have been enabled and have received cache fills, perform the following steps:

1. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.
2. Flush all caches using the WBINVD instruction.
3. Disable the MTRRs and set the default memory type to uncached or set all MTRRs for the uncached memory type (see the discussion of the discussion of the TYPE field and the E flag in Section 11.11.2.1, “IA32_MTRR_DEF_TYPE MSR”).

The caches must be flushed (step 2) after the CD flag is set to insure system memory coherency. If the caches are not flushed, cache hits on reads will still occur and data will be read from valid cache lines.

The intent of the three separate steps listed above address three distinct requirements: (i) discontinue new data replacing existing data in the cache (ii) ensure data already in the cache are evicted to memory, (iii) ensure subsequent memory references observe UC memory type semantics. Different processor implementation of caching control hardware may allow some variation of software implementation of these three requirements. See note below.

NOTES

Setting the CD flag in control register CR0 modifies the processor’s caching behaviour as indicated in Table 11-5, but setting the CD flag alone may not be sufficient across all processor families to force the effective memory type for all physical memory to be UC nor does it force strict memory ordering, due to hardware implementation variations across different processor families. To force the UC memory type and strict memory ordering on all of physical memory, it is sufficient to either program the MTRRs for all physical memory to be UC memory type or disable all MTRRs.

For the Pentium 4 and Intel Xeon processors, after the sequence of steps given above has been executed, the cache lines containing the code between the end of the WBINVD instruction and before the MTRRS have actually been disabled may be retained in the cache hierarchy. Here, to

remove code from the cache completely, a second WBINVD instruction must be executed after the MTRRs have been disabled.

For Intel Atom processors, setting the CD flag forces all physical memory to observe UC semantics (without requiring memory type of physical memory to be set explicitly). Consequently, software does not need to issue a second WBINVD as some other processor generations might require.

11.5.4 Disabling and Enabling the L3 Cache

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag (bit 6 of the IA32_MISC_ENABLE MSR) allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches. Prior to using this control to disable or enable the L3 cache, software should disable and flush all the processor caches, as described earlier in Section 11.5.3, “Preventing Caching,” to prevent loss of information stored in the L3 cache. After the L3 cache has been disabled or enabled, caching for the whole processor can be restored.

Newer Intel 64 processor with L3 do not support IA32_MISC_ENABLE[bit 6], the procedure described in Section 11.5.3, “Preventing Caching,” apply to the entire cache hierarchy.

11.5.5 Cache Management Instructions

The Intel 64 and IA-32 architectures provide several instructions for managing the L1, L2, and L3 caches. The INVD and WBINVD instructions are privileged instructions and operate on the L1, L2 and L3 caches as a whole. The PREFETCHh, CLFLUSH and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) offer more granular control over caching, and are available to all privileged levels.

The INVD and WBINVD instructions are used to invalidate the contents of the L1, L2, and L3 caches. The INVD instruction invalidates all internal cache entries, then generates a special-function bus cycle that indicates that external caches also should be invalidated. The INVD instruction should be used with care. It does not force a write-back of modified cache lines; therefore, data stored in the caches and not written back to system memory will be lost. Unless there is a specific requirement or benefit to invalidating the caches without writing back the modified lines (such as, during testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

The WBINVD instruction first writes back any modified lines in all the internal caches, then invalidates the contents of both the L1, L2, and L3 caches. It ensures that cache coherency with main memory is maintained regardless of the write policy in effect (that is, write-through or write-back). Following this operation, the WBINVD instruction generates one (P6 family processors) or two (Pentium and Intel486 processors) special-function bus cycles to indicate to external cache controllers that write-back of modified data followed by invalidation of external caches should occur. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.

The PREFETCHh instructions allow a program to suggest to the processor that a cache line from a specified location in system memory be prefetched into the cache hierarchy (see Section 11.8, “Explicit Caching”).

The CLFLUSH and CLFLUSHOPT instructions allow selected cache lines to be flushed from memory. These instructions give a program the ability to explicitly free up cache space, when it is known that cached section of system memory will not be accessed in the near future.

The non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) allow data to be moved from the processor’s registers directly into system memory without being also written into the L1, L2, and/or L3 caches. These instructions can be used to prevent cache pollution when operating on data that is going to be modified only once before being stored back into system memory. These instructions operate on data in the general-purpose, MMX, and XMM registers.

11.5.6 L1 Data Cache Context Mode

L1 data cache context mode is a feature of processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology. When `CPUID.1:ECX[bit 10] = 1`, the processor supports setting L1 data cache context mode using the L1 data cache context mode flag (`IA32_MISC_ENABLE[bit 24]`). Selectable modes are adaptive mode (default) and shared mode.

The BIOS is responsible for configuring the L1 data cache context mode.

11.5.6.1 Adaptive Mode

Adaptive mode facilitates L1 data cache sharing between logical processors. When running in adaptive mode, the L1 data cache is shared across logical processors in the same core if:

- CR3 control registers for logical processors sharing the cache are identical.
- The same paging mode is used by logical processors sharing the cache.

In this situation, the entire L1 data cache is available to each logical processor (instead of being competitively shared).

If CR3 values are different for the logical processors sharing an L1 data cache or the logical processors use different paging modes, processors compete for cache resources. This reduces the effective size of the cache for each logical processor. Aliasing of the cache is not allowed (which prevents data thrashing).

11.5.6.2 Shared Mode

In shared mode, the L1 data cache is competitively shared between logical processors. This is true even if the logical processors use identical CR3 registers and paging modes.

In shared mode, linear addresses in the L1 data cache can be aliased, meaning that one linear address in the cache can point to different physical locations. The mechanism for resolving aliasing can lead to thrashing. For this reason, `IA32_MISC_ENABLE[bit 24] = 0` is the preferred configuration for processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology.

11.6 SELF-MODIFYING CODE

A write to a memory location in a code segment that is currently cached in the processor causes the associated cache line (or lines) to be invalidated. This check is based on the physical address of the instruction. In addition, the P6 family and Pentium processors check whether a write to a code segment may modify an instruction that has been prefetched for execution. If the write affects a prefetched instruction, the prefetch queue is invalidated. This latter check is based on the linear address of the instruction. For the Pentium 4 and Intel Xeon processors, a write or a snoop of an instruction in a code segment, where the target instruction is already decoded and resident in the trace cache, invalidates the entire trace cache. The latter behavior means that programs that self-modify code can cause severe degradation of performance when run on the Pentium 4 and Intel Xeon processors.

In practice, the check on linear addresses should not create compatibility problems among IA-32 processors. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. Systems software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction, will execute a serializing operation, such as a `CPUID` instruction, before the modified instruction is executed, which will automatically resynchronize the instruction cache and prefetch queue. (See Section 8.1.3, "Handling Self- and Cross-Modifying Code," for more information about the use of self-modifying code.)

For Intel486 processors, a write to an instruction in the cache will modify it in both the cache and memory, but if the instruction was prefetched before the write, the old version of the instruction could be the one executed. To prevent the old instruction from being executed, flush the instruction prefetch unit by coding a jump instruction immediately after any write that modifies an instruction.

11.7 IMPLICIT CACHING (PENTIUM 4, INTEL XEON, AND P6 FAMILY PROCESSORS)

Implicit caching occurs when a memory element is made potentially cacheable, although the element may never have been accessed in the normal von Neumann sequence. Implicit caching occurs on the P6 and more recent processor families due to aggressive prefetching, branch prediction, and TLB miss handling. Implicit caching is an extension of the behavior of existing Intel386, Intel486, and Pentium processor systems, since software running on these processor families also has not been able to deterministically predict the behavior of instruction prefetch.

To avoid problems related to implicit caching, the operating system must explicitly invalidate the cache when changes are made to cacheable data that the cache coherency mechanism does not automatically handle. This includes writes to dual-ported or physically aliased memory boards that are not detected by the snooping mechanisms of the processor, and changes to page-table entries in memory.

The code in Example 11-1 shows the effect of implicit caching on page-table entries. The linear address F000H points to physical location B000H (the page-table entry for F000H contains the value B000H), and the page-table entry for linear address F000 is PTE_F000.

Example 11-1. Effect of Implicit Caching on Page-Table Entries

```
mov EAX, CR3; Invalidate the TLB
mov CR3, EAX; by copying CR3 to itself
mov PTE_F000, A000H; Change F000H to point to A000H
mov EBX, [F000H];
```

Because of speculative execution in the P6 and more recent processor families, the last MOV instruction performed would place the value at physical location B000H into EBX, rather than the value at the new physical address A000H. This situation is remedied by placing a TLB invalidation between the load and the store.

11.8 EXPLICIT CACHING

The Pentium III processor introduced four new instructions, the PREFETCH h instructions, that provide software with explicit control over the caching of data. These instructions provide “hints” to the processor that the data requested by a PREFETCH h instruction should be read into cache hierarchy now or as soon as possible, in anticipation of its use. The instructions provide different variations of the hint that allow selection of the cache level into which data will be read.

The PREFETCH h instructions can help reduce the long latency typically associated with reading data from memory and thus help prevent processor “stalls.” However, these instructions should be used judiciously. Overuse can lead to resource conflicts and hence reduce the performance of an application. Also, these instructions should only be used to prefetch data from memory; they should not be used to prefetch instructions. For more detailed information on the proper use of the prefetch instruction, refer to Chapter 7, “Optimizing Cache Usage,” in the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

11.9 INVALIDATING THE TRANSLATION LOOKASIDE BUFFERS (TLBS)

The processor updates its address translation caches (TLBs) transparently to software. Several mechanisms are available, however, that allow software and hardware to invalidate the TLBs either explicitly or as a side effect of another operation. Most details are given in Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches.” In addition, the following operations invalidate all TLB entries, irrespective of the setting of the G flag:

- Asserting or de-asserting the FLUSH# pin.
- (Pentium 4, Intel Xeon, and later processors only.) Writing to an MTRR (with a WRMSR instruction).
- Writing to control register CR0 to modify the PG or PE flag.

- (Pentium 4, Intel Xeon, and later processors only.) Writing to control register CR4 to modify the PSE, PGE, or PAE flag.
- Writing to control register CR4 to change the PCIDE flag from 1 to 0.

See Section 4.10, “Caching Translation Information,” for additional information about the TLBs.

11.10 STORE BUFFER

Intel 64 and IA-32 processors temporarily store each write (store) to memory in a store buffer. The store buffer improves processor performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or to a cache is complete. It also allows writes to be delayed for more efficient use of memory-access bus cycles.

In general, the existence of the store buffer is transparent to software, even in systems that use multiple processors. The processor ensures that write operations are always carried out in program order. It also insures that the contents of the store buffer are always drained to memory in the following situations:

- When an exception or interrupt is generated.
- (P6 and more recent processor families only) When a serializing instruction is executed.
- When an I/O instruction is executed.
- When a LOCK operation is performed.
- (P6 and more recent processor families only) When a BINIT operation is performed.
- (Pentium III, and more recent processor families only) When using an SFENCE instruction to order stores.
- (Pentium 4 and more recent processor families only) When using an MFENCE instruction to order stores.

The discussion of write ordering in Section 8.2, “Memory Ordering,” gives a detailed description of the operation of the store buffer.

11.11 MEMORY TYPE RANGE REGISTERS (MTRRS)

The following section pertains only to the P6 and more recent processor families.

The memory type range registers (MTRRs) provide a mechanism for associating the memory types (see Section 11.3, “Methods of Caching Available”) with physical-address ranges in system memory. They allow the processor to optimize operations for different types of memory such as RAM, ROM, frame-buffer memory, and memory-mapped I/O devices. They also simplify system hardware design by eliminating the memory control pins used for this function on earlier IA-32 processors and the external logic needed to drive them.

The MTRR mechanism allows **multiple** ranges to be defined in physical memory, and it defines a set of model-specific registers (MSRs) for specifying the type of memory that is contained in each range. Table 11-8 shows the memory types that can be specified and their properties; Figure 11-4 shows the mapping of physical memory with MTRRs. See Section 11.3, “Methods of Caching Available,” for a more detailed description of each memory type.

Following a hardware reset, the P6 and more recent processor families disable all the fixed and variable MTRRs, which in effect makes all of physical memory uncacheable. Initialization software should then set the MTRRs to a specific, system-defined memory map. Typically, the BIOS (basic input/output system) software configures the MTRRs. The operating system or executive is then free to modify the memory map using the normal page-level cacheability attributes.

In a multiprocessor system using a processor in the P6 family or a more recent family, each processor **MUST** use the identical MTRR memory map so that software will have a consistent view of memory.

NOTE

In multiple processor systems, the operating system must maintain MTRR consistency between all the processors in the system (that is, all processors must use the same MTRR values). The P6 and more recent processor families provide no hardware support for maintaining this consistency.

Table 11-8. Memory Types That Can Be Encoded in MTRRs

Memory Type and Mnemonic	Encoding in MTRR
Uncacheable (UC)	00H
Write Combining (WC)	01H
Reserved*	02H
Reserved*	03H
Write-through (WT)	04H
Write-protected (WP)	05H
Writeback (WB)	06H
Reserved*	7H through FFH

NOTE:

* Use of these encodings results in a general-protection exception (#GP).

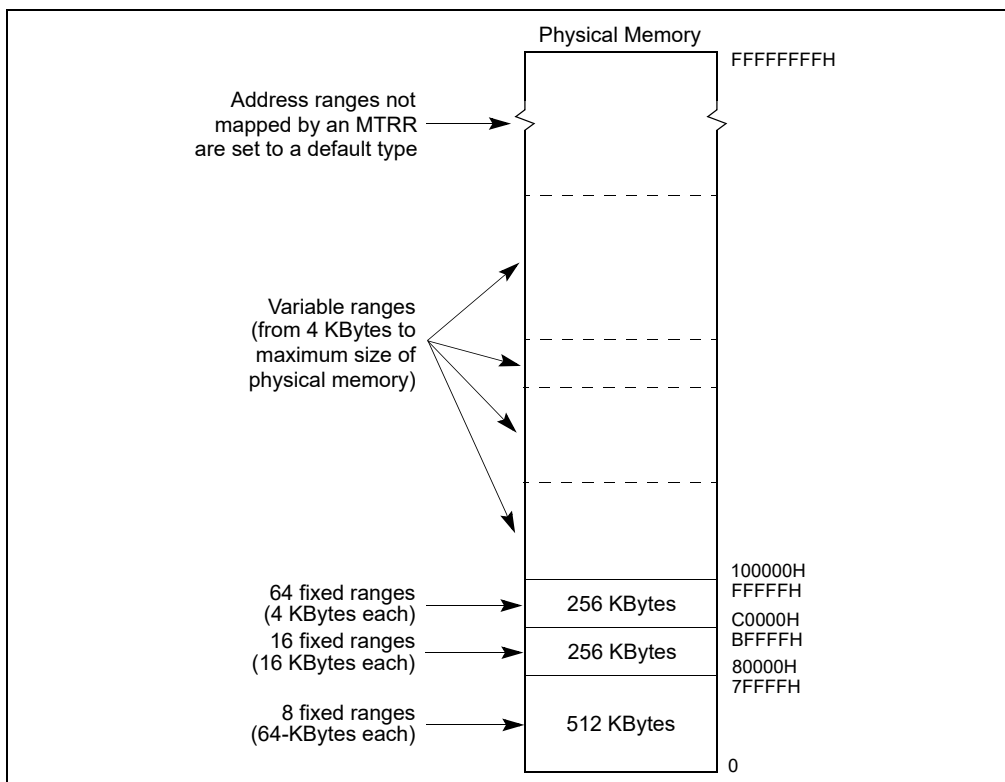


Figure 11-4. Mapping Physical Memory With MTRRs

11.11.1 MTRR Feature Identification

The availability of the MTRR feature is model-specific. Software can determine if MTRRs are supported on a processor by executing the CPUID instruction and reading the state of the MTRR flag (bit 12) in the feature information register (EDX).

If the MTRR flag is set (indicating that the processor implements MTRRs), additional information about MTRRs can be obtained from the 64-bit IA32_MTRRCAP MSR (named MTRRcap MSR for the P6 family processors). The IA32_MTRRCAP MSR is a read-only MSR that can be read with the RDMSR instruction. Figure 11-5 shows the contents of the IA32_MTRRCAP MSR. The functions of the flags and field in this register are as follows:

- **VCNT (variable range registers count) field, bits 0 through 7** — Indicates the number of variable ranges implemented on the processor.
- **FIX (fixed range registers supported) flag, bit 8** — Fixed range MTRRs (IA32_MTRR_FIX64K_00000 through IA32_MTRR_FIX4K_0F8000) are supported when set; no fixed range registers are supported when clear.
- **WC (write combining) flag, bit 10** — The write-combining (WC) memory type is supported when set; the WC type is not supported when clear.
- **SMRR (System-Management Range Register) flag, bit 11** — The system-management range register (SMRR) interface is supported when bit 11 is set; the SMRR interface is not supported when clear.

Bit 9 and bits 12 through 63 in the IA32_MTRRCAP MSR are reserved. If software attempts to write to the IA32_MTRRCAP MSR, a general-protection exception (#GP) is generated.

Software must read IA32_MTRRCAP VCNT field to determine the number of variable MTRRs and query other feature bits in IA32_MTRRCAP to determine additional capabilities that are supported in a processor. For example, some processors may report a value of '8' in the VCNT field, other processors may report VCNT with different values.

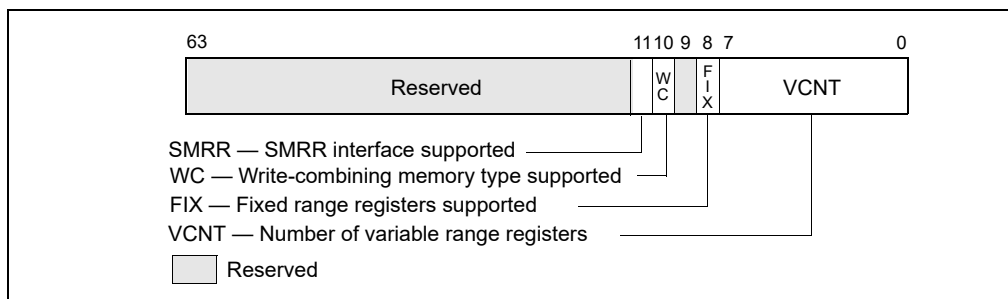


Figure 11-5. IA32_MTRRCAP Register

11.11.2 Setting Memory Ranges with MTRRs

The memory ranges and the types of memory specified in each range are set by three groups of registers: the IA32_MTRR_DEF_TYPE MSR, the fixed-range MTRRs, and the variable range MTRRs. These registers can be read and written to using the RDMSR and WRMSR instructions, respectively. The IA32_MTRRCAP MSR indicates the availability of these registers on the processor (see Section 11.11.1, “MTRR Feature Identification”).

11.11.2.1 IA32_MTRR_DEF_TYPE MSR

The IA32_MTRR_DEF_TYPE MSR (named MTRRdefType MSR for the P6 family processors) sets the default properties of the regions of physical memory that are not encompassed by MTRRs. The functions of the flags and field in this register are as follows:

- **Type field, bits 0 through 7** — Indicates the default memory type used for those physical memory address ranges that do not have a memory type specified for them by an MTRR (see Table 11-8 for the encoding of this field). The legal values for this field are 0, 1, 4, 5, and 6. All other values result in a general-protection exception (#GP) being generated.

Intel recommends the use of the UC (uncached) memory type for all physical memory addresses where memory does not exist. To assign the UC type to nonexistent memory locations, it can either be specified as the default type in the Type field or be explicitly assigned with the fixed and variable MTRRs.

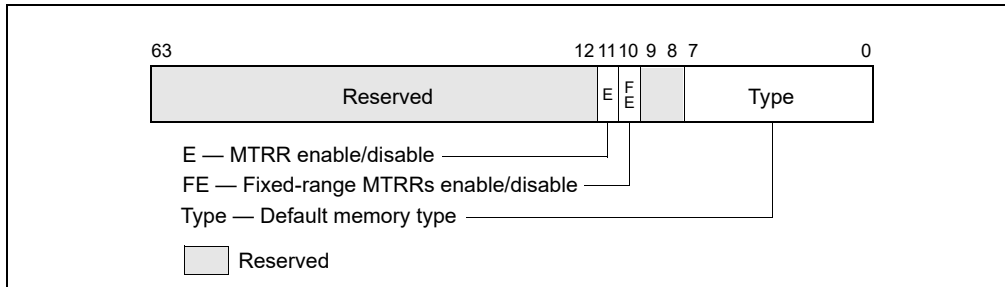


Figure 11-6. IA32_MTRR_DEF_TYPE MSR

- **FE (fixed MTRRs enabled) flag, bit 10** — Fixed-range MTRRs are enabled when set; fixed-range MTRRs are disabled when clear. When the fixed-range MTRRs are enabled, they take priority over the variable-range MTRRs when overlaps in ranges occur. If the fixed-range MTRRs are disabled, the variable-range MTRRs can still be used and can map the range ordinarily covered by the fixed-range MTRRs.
- **E (MTRRs enabled) flag, bit 11** — MTRRs are enabled when set; all MTRRs are disabled when clear, and the UC memory type is applied to all of physical memory. When this flag is set, the FE flag can disable the fixed-range MTRRs; when the flag is clear, the FE flag has no affect. When the E flag is set, the type specified in the default memory type field is used for areas of memory not already mapped by either a fixed or variable MTRR.

Bits 8 and 9, and bits 12 through 63, in the IA32_MTRR_DEF_TYPE MSR are reserved; the processor generates a general-protection exception (#GP) if software attempts to write nonzero values to them.

11.11.2.2 Fixed Range MTRRs

The fixed memory ranges are mapped with 11 fixed-range registers of 64 bits each. Each of these registers is divided into 8-bit fields that are used to specify the memory type for each of the sub-ranges the register controls:

- **Register IA32_MTRR_FIX64K_00000** — Maps the 512-KByte address range from 0H to 7FFFFH. This range is divided into eight 64-KByte sub-ranges.
- **Registers IA32_MTRR_FIX16K_80000 and IA32_MTRR_FIX16K_A0000** — Maps the two 128-KByte address ranges from 80000H to BFFFFH. This range is divided into sixteen 16-KByte sub-ranges, 8 ranges per register.
- **Registers IA32_MTRR_FIX4K_C0000 through IA32_MTRR_FIX4K_F8000** — Maps eight 32-KByte address ranges from C0000H to FFFFFH. This range is divided into sixty-four 4-KByte sub-ranges, 8 ranges per register.

Table 11-9 shows the relationship between the fixed physical-address ranges and the corresponding fields of the fixed-range MTRRs; Table 11-8 shows memory type encoding for MTRRs.

For the P6 family processors, the prefix for the fixed range MTRRs is MTRRfix.

11.11.2.3 Variable Range MTRRs

The Pentium 4, Intel Xeon, and P6 family processors permit software to specify the memory type for m variable-size address ranges, using a pair of MTRRs for each range. The number m of ranges supported is given in bits 7:0 of the IA32_MTRRCAP MSR (see Figure 11-5 in Section 11.11.1).

The first entry in each pair (IA32_MTRR_PHYSBASE n) defines the base address and memory type for the range; the second entry (IA32_MTRR_PHYSMASK n) contains a mask used to determine the address range. The “ n ” suffix is in the range 0 through $m-1$ and identifies a specific register pair.

For P6 family processors, the prefixes for these variable range MTRRs are MTRRphysBase and MTRRphysMask.

Table 11-9. Address Mapping for Fixed-Range MTRRs

Address Range (hexadecimal)								MTRR
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
7000-7FFFF	6000-6FFFF	5000-5FFFF	4000-4FFFF	3000-3FFFF	2000-2FFFF	1000-1FFFF	0000-0FFFF	IA32_MTRR_FIX64K_00000
9C000-9FFFF	98000-9BFFF	94000-97FFF	90000-93FFF	8C000-8FFFF	88000-8BFFF	84000-87FFF	80000-83FFF	IA32_MTRR_FIX16K_80000
BC000-BFFFF	B8000-BBFFF	B4000-B7FFF	B0000-B3FFF	AC000-AFFFF	A8000-ABFFF	A4000-A7FFF	A0000-A3FFF	IA32_MTRR_FIX16K_A0000
C7000-C7FFF	C6000-C6FFF	C5000-C5FFF	C4000-C4FFF	C3000-C3FFF	C2000-C2FFF	C1000-C1FFF	C0000-C0FFF	IA32_MTRR_FIX4K_C0000
CF000-CFFFF	CE000-CEFFF	CD000-CDFFF	CC000-CCFFF	CB000-CBFFF	CA000-CAFFF	C9000-C9FFF	C8000-C8FFF	IA32_MTRR_FIX4K_C8000
D7000-D7FFF	D6000-D6FFF	D5000-D5FFF	D4000-D4FFF	D3000-D3FFF	D2000-D2FFF	D1000-D1FFF	D0000-D0FFF	IA32_MTRR_FIX4K_D0000
DF000-DFFFF	DE000-DEFFF	DD000-DDFFF	DC000-DCFFF	DB000-DBFFF	DA000-DAFFF	D9000-D9FFF	D8000-D8FFF	IA32_MTRR_FIX4K_D8000
E7000-E7FFF	E6000-E6FFF	E5000-E5FFF	E4000-E4FFF	E3000-E3FFF	E2000-E2FFF	E1000-E1FFF	E0000-E0FFF	IA32_MTRR_FIX4K_E0000
EF000-EFFFF	EE000-EEFFF	ED000-EDFFF	EC000-ECFFF	EB000-EBFFF	EA000-EAFFF	E9000-E9FFF	E8000-E8FFF	IA32_MTRR_FIX4K_E8000
F7000-F7FFF	F6000-F6FFF	F5000-F5FFF	F4000-F4FFF	F3000-F3FFF	F2000-F2FFF	F1000-F1FFF	F0000-F0FFF	IA32_MTRR_FIX4K_F0000
FF000-FFFFF	FE000-FEFFF	FD000-FDFFF	FC000-FCFFF	FB000-FBFFF	FA000-FAFFF	F9000-F9FFF	F8000-F8FFF	IA32_MTRR_FIX4K_F8000

Figure 11-7 shows flags and fields in these registers. The functions of these flags and fields are:

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 11-8 for the encoding of this field).
- **PhysBase field, bits 12 through (MAXPHYADDR-1)** — Specifies the base address of the address range. This 24-bit value, in the case where MAXPHYADDR is 36 bits, is extended by 12 bits at the low end to form the base address (this automatically aligns the address on a 4-KByte boundary).
- **PhysMask field, bits 12 through (MAXPHYADDR-1)** — Specifies a mask (24 bits if the maximum physical address size is 36 bits, 28 bits if the maximum physical address size is 40 bits). The mask determines the range of the region being mapped, according to the following relationships:
 - Address_Within_Range AND PhysMask = PhysBase AND PhysMask
 - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 11.11.3, “Example Base and Mask Calculations.”
 - The width of the PhysMask field depends on the maximum physical address size supported by the processor.

CPUID.80000008H reports the maximum physical address size supported by the processor. If CPUID.80000008H is not available, software may assume that the processor supports a 36-bit physical address size (then PhysMask is 24 bits wide and the upper 28 bits of IA32_MTRR_PHYSMASKn are reserved). See the Note below.
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

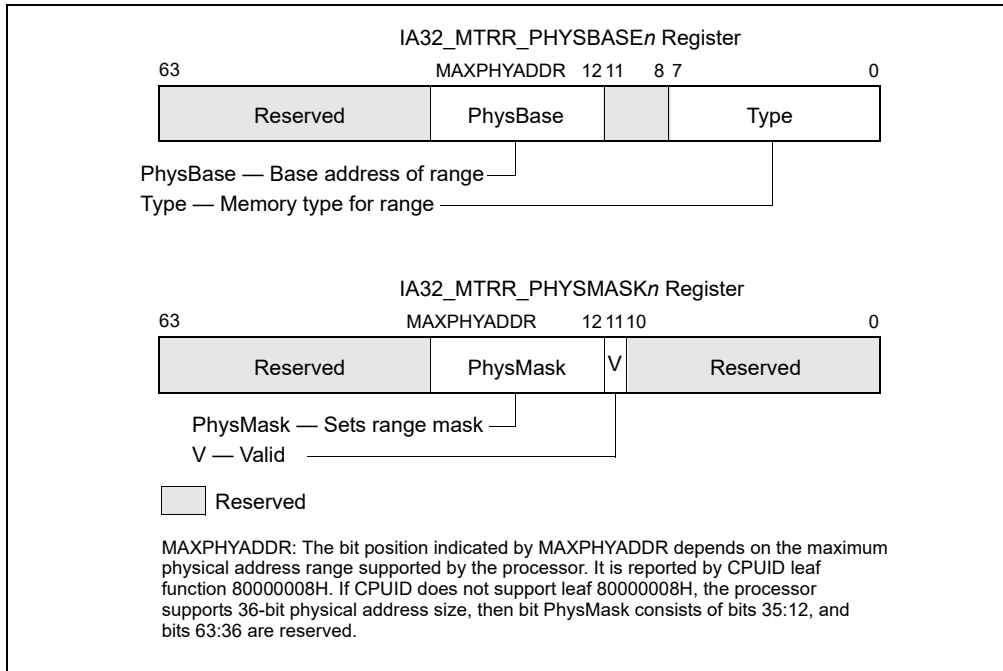


Figure 11-7. IA32_MTRR_PHYSBASE_n and IA32_MTRR_PHYSMASK_n Variable-Range Register Pair

All other bits in the IA32_MTRR_PHYSBASE_n and IA32_MTRR_PHYSMASK_n registers are reserved; the processor generates a general-protection exception (#GP) if software attempts to write to them.

Some mask values can result in ranges that are not continuous. In such ranges, the area not mapped by the mask value is set to the default memory type, unless some other MTRR specifies a type for that range. Intel does not encourage the use of “discontinuous” ranges.

NOTE

It is possible for software to parse the memory descriptions that BIOS provides by using the ACPI/INT15 e820 interface mechanism. This information then can be used to determine how MTRRs are initialized (for example: allowing the BIOS to define valid memory ranges and the maximum memory range supported by the platform, including the processor).

See Section 11.11.4.1, “MTRR Precedences,” for information on overlapping variable MTRR ranges.

11.11.2.4 System-Management Range Register Interface

If IA32_MTRRCAP[bit 11] is set, the processor supports the SMRR interface to restrict access to a specified memory address range used by system-management mode (SMM) software (see Section 34.4.2.1). If the SMRR interface is supported, SMM software is strongly encouraged to use it to protect the SMI code and data stored by SMI handler in the SMRAM region.

The system-management range registers consist of a pair of MSRs (see Figure 11-8). The IA32_SMRR_PHYSBASE MSR defines the base address for the SMRAM memory range and the memory type used to access it in SMM. The IA32_SMRR_PHYSMASK MSR contains a valid bit and a mask that determines the SMRAM address range protected by the SMRR interface. These MSRs may be written only in SMM; an attempt to write them outside of SMM causes a general-protection exception.¹

Figure 11-8 shows flags and fields in these registers. The functions of these flags and fields are the following:

1. For some processor models, these MSRs can be accessed by RDMSR and WRMSR only if the SMRR interface has been enabled using a model-specific bit in the IA32_FEATURE_CONTROL MSR.

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 11-8 for the encoding of this field).
- **PhysBase field, bits 12 through 31** — Specifies the base address of the address range. The address must be less than 4 GBytes and is automatically aligned on a 4-KByte boundary.
- **PhysMask field, bits 12 through 31** — Specifies a mask that determines the range of the region being mapped, according to the following relationships:
 - $\text{Address_Within_Range AND PhysMask} = \text{PhysBase AND PhysMask}$
 - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 11.11.3, “Example Base and Mask Calculations.”
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

Before attempting to access these SMRR registers, software must test bit 11 in the IA32_MTRRCAP register. If SMRR is not supported, reads from or writes to registers cause general-protection exceptions.

When the valid flag in the IA32_SMRR_PHYSMASK MSR is 1, accesses to the specified address range are treated as follows:

- If the logical processor is in SMM, accesses uses the memory type in the IA32_SMRR_PHYSBASE MSR.
- If the logical processor is not in SMM, write accesses are ignored and read accesses return a fixed value for each byte. The uncacheable memory type (UC) is used in this case.

The above items apply even if the address range specified overlaps with a range specified by the MTRRs.

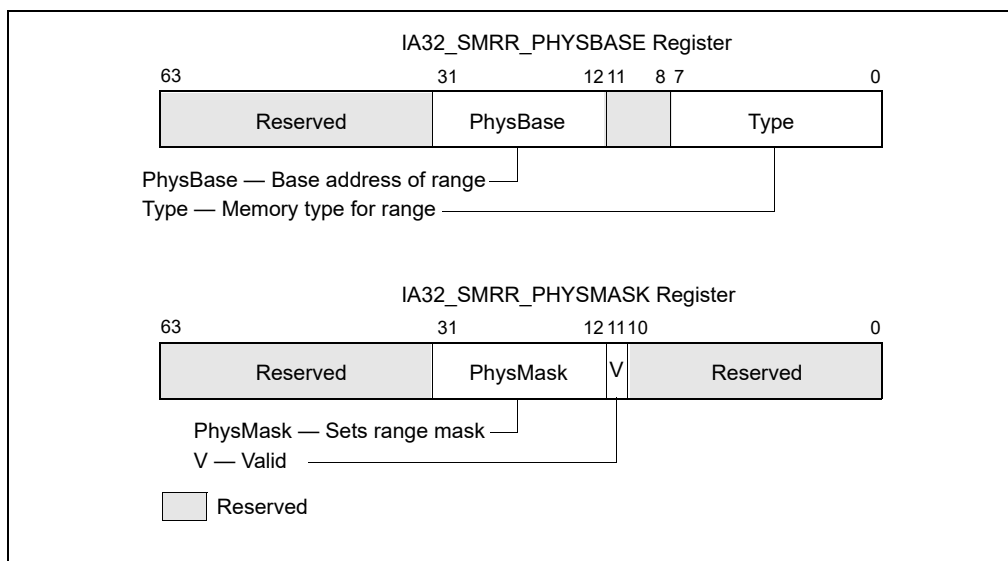


Figure 11-8. IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK SMRR Pair

11.11.3 Example Base and Mask Calculations

The examples in this section apply to processors that support a maximum physical address size of 36 bits. The base and mask values entered in variable-range MTRR pairs are 24-bit values that the processor extends to 36-bits.

For example, to enter a base address of 2 MBytes (200000H) in the IA32_MTRR_PHYSBASE3 register, the 12 least-significant bits are truncated and the value 000200H is entered in the PhysBase field. The same operation must be performed on mask values. For example, to map the address range from 200000H to 3FFFFFFH (2 MBytes to 4 MBytes), a mask value of FFFE0000H is required. Again, the 12 least-significant bits of this mask value are truncated, so that the value entered in the PhysMask field of IA32_MTRR_PHYSMASK3 is FFFE00H. This mask is chosen so that when any address in the 200000H to 3FFFFFFH range is AND'd with the mask value, it will return the same value as when the base address is AND'd with the mask value (which is 200000H).

To map the address range from 400000H to 7FFFFFFH (4 MBytes to 8 MBytes), a base value of 000400H is entered in the PhysBase field and a mask value of FFFC00H is entered in the PhysMask field.

Example 11-2. Setting-Up Memory for a System

Here is an example of setting up the MTRRs for an system. Assume that the system has the following characteristics:

- 96 MBytes of system memory is mapped as write-back memory (WB) for highest system performance.
- A custom 4-MByte I/O card is mapped to uncached memory (UC) at a base address of 64 MBytes. This restriction forces the 96 MBytes of system memory to be addressed from 0 to 64 MBytes and from 68 MBytes to 100 MBytes, leaving a 4-MByte hole for the I/O card.
- An 8-MByte graphics card is mapped to write-combining memory (WC) beginning at address A0000000H.
- The BIOS area from 15 MBytes to 16 MBytes is mapped to UC memory.

The following settings for the MTRRs will yield the proper mapping of the physical address space for this system configuration.

```
IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 000F FC00 0800H
Caches 0-64 MByte as WB cache type.
```

```
IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 000F FE00 0800H
Caches 64-96 MByte as WB cache type.
```

```
IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 000F FFC0 0800H
Caches 96-100 MByte as WB cache type.
```

```
IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 000F FFC0 0800H
Caches 64-68 MByte as UC cache type.
```

```
IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 000F FFF0 0800H
Caches 15-16 MByte as UC cache type.
```

```
IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 000F FF80 0800H
Caches A0000000-A0800000 as WC type.
```

This MTRR setup uses the ability to overlap any two memory ranges (as long as the ranges are mapped to WB and UC memory types) to minimize the number of MTRR registers that are required to configure the memory environment. This setup also fulfills the requirement that two register pairs are left for operating system usage.

11.11.3.1 Base and Mask Calculations for Greater-Than 36-bit Physical Address Support

For Intel 64 and IA-32 processors that support greater than 36 bits of physical address size, software should query CPUID.80000008H to determine the maximum physical address. See the example.

Example 11-3. Setting-Up Memory for a System with a 40-Bit Address Size

If a processor supports 40-bits of physical address size, then the PhysMask field (in IA32_MTRR_PHYSMASK_n registers) is 28 bits instead of 24 bits. For this situation, Example 11-2 should be modified as follows:

```
IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 00FF FC00 0800H
Caches 0-64 MByte as WB cache type.
```

MEMORY CACHE CONTROL

IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 00FF FE00 0800H
Caches 64-96 MByte as WB cache type.

IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 00FF FFC0 0800H
Caches 96-100 MByte as WB cache type.

IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 00FF FFC0 0800H
Caches 64-68 MByte as UC cache type.

IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 00FF FFF0 0800H
Caches 15-16 MByte as UC cache type.

IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 00FF FF80 0800H
Caches A0000000-A0800000 as WC type.

11.11.4 Range Size and Alignment Requirement

A range that is to be mapped to a variable-range MTRR must meet the following “power of 2” size and alignment rules:

1. The minimum range size is 4 KBytes and the base address of the range must be on at least a 4-KByte boundary.
2. For ranges greater than 4 KBytes, each range must be of length 2^n and its base address must be aligned on a 2^n boundary, where n is a value equal to or greater than 12. The base-address alignment value cannot be less than its length. For example, an 8-KByte range cannot be aligned on a 4-KByte boundary. It must be aligned on at least an 8-KByte boundary.

11.11.4.1 MTRR Precedences

If the MTRRs are not enabled (by setting the E flag in the IA32_MTRR_DEF_TYPE MSR), then all memory accesses are of the UC memory type. If the MTRRs are enabled, then the memory type used for a memory access is determined as follows:

1. If the physical address falls within the first 1 MByte of physical memory and fixed MTRRs are enabled, the processor uses the memory type stored for the appropriate fixed-range MTRR.
2. Otherwise, the processor attempts to match the physical address with a memory type set by the variable-range MTRRs:
 - If one variable memory range matches, the processor uses the memory type stored in the IA32_MTRR_PHYSBASE n register for that range.
 - If two or more variable memory ranges match and the memory types are identical, then that memory type is used.
 - If two or more variable memory ranges match and one of the memory types is UC, the UC memory type is used.
 - If two or more variable memory ranges match and the memory types are WT and WB, the WT memory type is used.
 - For overlaps not defined by the above rules, processor behavior is undefined.
3. If no fixed or variable memory range matches, the processor uses the default memory type.

11.11.5 MTRR Initialization

On a hardware reset, the P6 and more recent processors clear the valid flags in variable-range MTRRs and clear the E flag in the IA32_MTRR_DEF_TYPE MSR to disable all MTRRs. All other bits in the MTRRs are undefined.

Prior to initializing the MTRRs, software (normally the system BIOS) must initialize all fixed-range and variable-range MTRR register fields to 0. Software can then initialize the MTRRs according to known types of memory, including memory on devices that it auto-configures. Initialization is expected to occur prior to booting the operating system.

See Section 11.11.8, “MTRR Considerations in MP Systems,” for information on initializing MTRRs in MP (multiple-processor) systems.

11.11.6 Remapping Memory Types

A system designer may re-map memory types to tune performance or because a future processor may not implement all memory types supported by the Pentium 4, Intel Xeon, and P6 family processors. The following rules support coherent memory-type re-mappings:

1. A memory type should not be mapped into another memory type that has a weaker memory ordering model. For example, the uncacheable type cannot be mapped into any other type, and the write-back, write-through, and write-protected types cannot be mapped into the weakly ordered write-combining type.
2. A memory type that does not delay writes should not be mapped into a memory type that does delay writes, because applications of such a memory type may rely on its write-through behavior. Accordingly, the write-back type cannot be mapped into the write-through type.
3. A memory type that views write data as not necessarily stored and read back by a subsequent read, such as the write-protected type, can only be mapped to another type with the same behaviour (and there are no others for the Pentium 4, Intel Xeon, and P6 family processors) or to the uncacheable type.

In many specific cases, a system designer can have additional information about how a memory type is used, allowing additional mappings. For example, write-through memory with no associated write side effects can be mapped into write-back memory.

11.11.7 MTRR Maintenance Programming Interface

The operating system maintains the MTRRs after booting and sets up or changes the memory types for memory-mapped devices. The operating system should provide a driver and application programming interface (API) to access and set the MTRRs. The function calls MemTypeGet() and MemTypeSet() define this interface.

11.11.7.1 MemTypeGet() Function

The MemTypeGet() function returns the memory type of the physical memory range specified by the parameters base and size. The base address is the starting physical address and the size is the number of bytes for the memory range. The function automatically aligns the base address and size to 4-KByte boundaries. Pseudocode for the MemTypeGet() function is given in Example 11-4.

Example 11-4. MemTypeGet() Pseudocode

```

#define MIXED_TYPES -1 /* 0 < MIXED_TYPES || MIXED_TYPES > 256 */

IF CPU_FEATURES.MTRR /* processor supports MTRRs */
    THEN
        Align BASE and SIZE to 4-KByte boundary;
        IF (BASE + SIZE) wrap physical-address space
            THEN return INVALID;
        FI;
        IF MTRRdefType.E = 0
            THEN return UC;
        FI;
        FirstType := Get4KMemType (BASE);
        /* Obtains memory type for first 4-KByte range. */
        /* See Get4KMemType (4KByteRange) in Example 11-5. */
        FOR each additional 4-KByte range specified in SIZE
            NextType := Get4KMemType (4KByteRange);
            IF NextType != FirstType
                THEN return Mixed_Types;
            FI;
        ROF;
        return FirstType;
    ELSE return UNSUPPORTED;
FI;

```

If the processor does not support MTRRs, the function returns UNSUPPORTED. If the MTRRs are not enabled, then the UC memory type is returned. If more than one memory type corresponds to the specified range, a status of MIXED_TYPES is returned. Otherwise, the memory type defined for the range (UC, WC, WT, WB, or WP) is returned.

The pseudocode for the Get4KMemType() function in Example 11-5 obtains the memory type for a single 4-KByte range at a given physical address. The sample code determines whether a PHY_ADDRESS falls within a fixed range by comparing the address with the known fixed ranges: 0 to 7FFFFH (64-KByte regions), 80000H to BFFFFH (16-KByte regions), and C0000H to FFFFFH (4-KByte regions). If an address falls within one of these ranges, the appropriate bits within one of its MTRRs determine the memory type.

Example 11-5. Get4KMemType() Pseudocode

```

IF IA32_MTRRCAP.FIX AND MTRRdefType.FE /* fixed registers enabled */
    THEN IF PHY_ADDRESS is within a fixed range
        return IA32_MTRR_FIX.Type;
    FI;
FOR each variable-range MTRR in IA32_MTRRCAP.VCNT
    IF IA32_MTRR_PHYSMASK.V = 0
        THEN continue;
    FI;
    IF (PHY_ADDRESS AND IA32_MTRR_PHYSMASK.Mask) =
        (IA32_MTRR_PHYSBASE.Base
        AND IA32_MTRR_PHYSMASK.Mask)
        THEN
            return IA32_MTRR_PHYSBASE.Type;
    FI;
ROF;
return MTRRdefType.Type;

```

11.11.7.2 MemTypeSet() Function

The MemTypeSet() function in Example 11-6 sets a MTRR for the physical memory range specified by the parameters base and size to the type specified by type. The base address and size are multiples of 4 KBytes and the size is not 0.

Example 11-6. MemTypeSet Pseudocode

```

IF CPU_FEATURES.MTRR (* processor supports MTRRs *)
  THEN
    IF BASE and SIZE are not 4-KByte aligned or size is 0
      THEN return INVALID;
    FI;
    IF (BASE + SIZE) wrap 4-GByte address space
      THEN return INVALID;
    FI;
    IF TYPE is invalid for Pentium 4, Intel Xeon, and P6 family
    processors
      THEN return UNSUPPORTED;
    FI;
    IF TYPE is WC and not supported
      THEN return UNSUPPORTED;
    FI;
    IF IA32_MTRRCAP.FIX is set AND range can be mapped using a
    fixed-range MTRR
      THEN
        pre_mtrr_change();
        update affected MTRR;
        post_mtrr_change();
      FI;

  ELSE (* try to map using a variable MTRR pair *)
    IF IA32_MTRRCAP.VCNT = 0
      THEN return UNSUPPORTED;
    FI;
    IF conflicts with current variable ranges
      THEN return RANGE_OVERLAP;
    FI;
    IF no MTRRs available
      THEN return VAR_NOT_AVAILABLE;
    FI;
    IF BASE and SIZE do not meet the power of 2 requirements for
    variable MTRRs
      THEN return INVALID_VAR_REQUEST;
    FI;
    pre_mtrr_change();
    Update affected MTRRs;
    post_mtrr_change();
  FI;

pre_mtrr_change()
  BEGIN
    disable interrupts;
    Save current value of CR4;
    disable and flush caches;
  
```


MEMORY CACHE CONTROL

```
flush TLBs;
disable MTRRs;
IF multiprocessing
    THEN maintain consistency through IPIs;
FI;
END
post_mtrr_change()
BEGIN
flush caches and TLBs;
enable MTRRs;
enable caches;
restore value of CR4;
enable interrupts;
END
```

The physical address to variable range mapping algorithm in the MemTypeSet function detects conflicts with current variable range registers by cycling through them and determining whether the physical address in question matches any of the current ranges. During this scan, the algorithm can detect whether any current variable ranges overlap and can be concatenated into a single range.

The pre_mtrr_change() function disables interrupts prior to changing the MTRRs, to avoid executing code with a partially valid MTRR setup. The algorithm disables caching by setting the CD flag and clearing the NW flag in control register CR0. The caches are invalidated using the WBINVD instruction. The algorithm flushes all TLB entries either by clearing the page-global enable (PGE) flag in control register CR4 (if PGE was already set) or by updating control register CR3 (if PGE was already clear). Finally, it disables MTRRs by clearing the E flag in the IA32_MTRR_DEF_TYPE MSR.

After the memory type is updated, the post_mtrr_change() function re-enables the MTRRs and again invalidates the caches and TLBs. This second invalidation is required because of the processor's aggressive prefetch of both instructions and data. The algorithm restores interrupts and re-enables caching by setting the CD flag.

An operating system can batch multiple MTRR updates so that only a single pair of cache invalidations occur.

11.11.8 MTRR Considerations in MP Systems

In MP (multiple-processor) systems, the operating systems must maintain MTRR consistency between all the processors in the system. The Pentium 4, Intel Xeon, and P6 family processors provide no hardware support to maintain this consistency. In general, all processors must have the same MTRR values.

This requirement implies that when the operating system initializes an MP system, it must load the MTRRs of the boot processor while the E flag in register MTRRdefType is 0. The operating system then directs other processors to load their MTRRs with the same memory map. After all the processors have loaded their MTRRs, the operating system signals them to enable their MTRRs. Barrier synchronization is used to prevent further memory accesses until all processors indicate that the MTRRs are enabled. This synchronization is likely to be a shoot-down style algorithm, with shared variables and interprocessor interrupts.

Any change to the value of the MTRRs in an MP system requires the operating system to repeat the loading and enabling process to maintain consistency, using the following procedure:

1. Broadcast to all processors to execute the following code sequence.
2. Disable interrupts.
3. Wait for all processors to reach this point.
4. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.)
5. Flush all caches using the WBINVD instructions. Note on a processor that supports self-snooping, CPUID feature flag bit 27, this step is unnecessary.
6. If the PGE flag is set in control register CR4, flush all TLBs by clearing that flag.

7. If the PGE flag is clear in control register CR4, flush all TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.
8. Disable all range registers (by clearing the E flag in register MTRRdefType). If only variable ranges are being modified, software may clear the valid bits for the affected register pairs instead.
9. Update the MTRRs.
10. Enable all range registers (by setting the E flag in register MTRRdefType). If only variable-range registers were modified and their individual valid bits were cleared, then set the valid bits for the affected ranges instead.
11. Flush all caches and all TLBs a second time. (The TLB flush is required for Pentium 4, Intel Xeon, and P6 family processors. Executing the WBINVD instruction is not needed when using Pentium 4, Intel Xeon, and P6 family processors, but it may be needed in future systems.)
12. Enter the normal cache mode to re-enable caching. (Set the CD and NW flags in control register CR0 to 0.)
13. Set PGE flag in control register CR4, if cleared in Step 6 (above).
14. Wait for all processors to reach this point.
15. Enable interrupts.

11.11.9 Large Page Size Considerations

The MTRRs provide memory typing for a limited number of regions that have a 4 KByte granularity (the same granularity as 4-KByte pages). The memory type for a given page is cached in the processor's TLBs. When using large pages (2 MBytes, 4 MBytes, or 1 GBytes), a single page-table entry covers multiple 4-KByte granules, each with a single memory type. Because the memory type for a large page is cached in the TLB, the processor can behave in an undefined manner if a large page is mapped to a region of memory that MTRRs have mapped with multiple memory types.

Undefined behavior can be avoided by insuring that all MTRR memory-type ranges within a large page are of the same type. If a large page maps to a region of memory containing different MTRR-defined memory types, the PCD and PWT flags in the page-table entry should be set for the most conservative memory type for that range. For example, a large page used for memory mapped I/O and regular memory is mapped as UC memory. Alternatively, the operating system can map the region using multiple 4-KByte pages each with its own memory type.

The requirement that all 4-KByte ranges in a large page are of the same memory type implies that large pages with different memory types may suffer a performance penalty, since they must be marked with the lowest common denominator memory type. The same consideration apply to 1 GByte pages, each of which may consist of multiple 2-Mbyte ranges.

The Pentium 4, Intel Xeon, and P6 family processors provide special support for the physical memory range from 0 to 4 MBytes, which is potentially mapped by both the fixed and variable MTRRs. This support is invoked when a Pentium 4, Intel Xeon, or P6 family processor detects a large page overlapping the first 1 MByte of this memory range with a memory type that conflicts with the fixed MTRRs. Here, the processor maps the memory range as multiple 4-KByte pages within the TLB. This operation insures correct behavior at the cost of performance. To avoid this performance penalty, operating-system software should reserve the large page option for regions of memory at addresses greater than or equal to 4 MBytes.

11.12 PAGE ATTRIBUTE TABLE (PAT)

The Page Attribute Table (PAT) extends the IA-32 architecture's page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings. The PAT is a companion feature to the MTRRs; that is, the MTRRs allow mapping of memory types to regions of the physical address space, where the PAT allows mapping of memory types to pages within the linear address space. The MTRRs are useful for statically describing memory types for physical ranges, and are typically set up by the system BIOS. The PAT extends the functions of the PCD and PWT bits in page tables to allow all five of the memory types that can be assigned with the MTRRs (plus one additional memory type) to also be assigned dynamically to pages of the linear address space.

The PAT was introduced to IA-32 architecture on the Pentium III processor. It is also available in the Pentium 4 and Intel Xeon processors.

11.12.1 Detecting Support for the PAT Feature

An operating system or executive can detect the availability of the PAT by executing the CPUID instruction with a value of 1 in the EAX register. Support for the PAT is indicated by the PAT flag (bit 16 of the values returned to EDX register). If the PAT is supported, the operating system or executive can use the IA32_PAT MSR to program the PAT. When memory types have been assigned to entries in the PAT, software can then use of the PAT-index bit (PAT) in the page-table and page-directory entries along with the PCD and PWT bits to assign memory types from the PAT to individual pages.

Note that there is no separate flag or control bit in any of the control registers that enables the PAT. The PAT is always enabled on all processors that support it, and the table lookup always occurs whenever paging is enabled, in all paging modes.

11.12.2 IA32_PAT MSR

The IA32_PAT MSR is located at MSR address 277H (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*). Figure 11-9. shows the format of the 64-bit IA32_PAT MSR.

The IA32_PAT MSR contains eight page attribute fields: PA0 through PA7. The three low-order bits of each field are used to specify a memory type. The five high-order bits of each field are reserved, and must be set to all 0s. Each of the eight page attribute fields can contain any of the memory type encodings specified in Table 11-10.

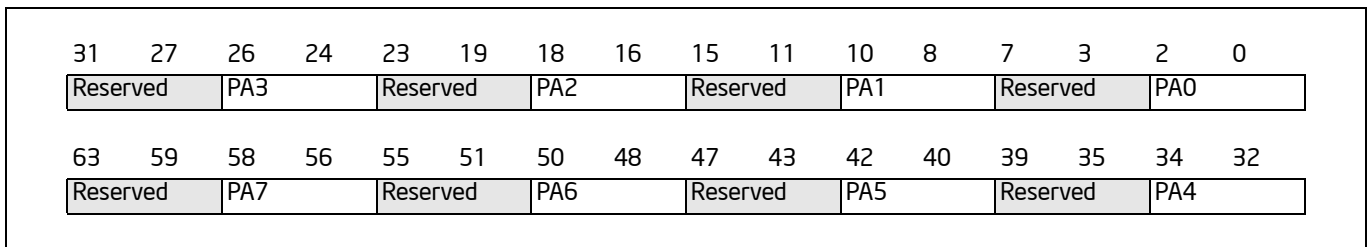


Figure 11-9. IA32_PAT MSR

Note that for the P6 family processors, the IA32_PAT MSR is named the PAT MSR.

Table 11-10. Memory Types That Can Be Encoded With PAT

Encoding	Mnemonic
00H	Uncacheable (UC)
01H	Write Combining (WC)
02H	Reserved*
03H	Reserved*
04H	Write Through (WT)
05H	Write Protected (WP)
06H	Write Back (WB)
07H	Uncached (UC-)
08H - FFH	Reserved*

NOTE:

* Using these encodings will result in a general-protection exception (#GP).

11.12.3 Selecting a Memory Type from the PAT

To select a memory type for a page from the PAT, a 3-bit index made up of the PAT, PCD, and PWT bits must be encoded in the page-table or page-directory entry for the page. Table 11-11 shows the possible encodings of the PAT, PCD, and PWT bits and the PAT entry selected with each encoding. The PAT bit is bit 7 in page-table entries that point to 4-KByte pages and bit 12 in paging-structure entries that point to larger pages. The PCD and PWT bits are bits 4 and 3, respectively, in paging-structure entries that point to pages of any size.

The PAT entry selected for a page is used in conjunction with the MTRR setting for the region of physical memory in which the page is mapped to determine the effective memory type for the page, as shown in Table 11-7.

Table 11-11. Selection of PAT Entries with PAT, PCD, and PWT Flags

PAT	PCD	PWT	PAT Entry
0	0	0	PAT0
0	0	1	PAT1
0	1	0	PAT2
0	1	1	PAT3
1	0	0	PAT4
1	0	1	PAT5
1	1	0	PAT6
1	1	1	PAT7

11.12.4 Programming the PAT

Table 11-12 shows the default setting for each PAT entry following a power up or reset of the processor. The setting remain unchanged following a soft reset (INIT reset).

Table 11-12. Memory Type Setting of PAT Entries Following a Power-up or Reset

PAT Entry	Memory Type Following Power-up or Reset
PAT0	WB
PAT1	WT
PAT2	UC-
PAT3	UC
PAT4	WB
PAT5	WT
PAT6	UC-
PAT7	UC

The values in all the entries of the PAT can be changed by writing to the IA32_PAT MSR using the WRMSR instruction. The IA32_PAT MSR is read and write accessible (use of the RDMSR and WRMSR instructions, respectively) to software operating at a CPL of 0. Table 11-10 shows the allowable encoding of the entries in the PAT. Attempting to write an undefined memory type encoding into the PAT causes a general-protection (#GP) exception to be generated.

The operating system is responsible for insuring that changes to a PAT entry occur in a manner that maintains the consistency of the processor caches and translation lookaside buffers (TLB). This is accomplished by following the procedure as specified in Section 11.11.8, "MTRR Considerations in MP Systems," for changing the value of an MTRR in a multiple processor system. It requires a specific sequence of operations that includes flushing the processors caches and TLBs.

The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical page mapped to two or more different linear addresses, each with different memory types. Intel does not support this practice because it may lead to undefined operations that can result in a system failure. In particular, a WC page must never be aliased to a cacheable page because WC writes may not check the processor caches.

When remapping a page that was previously mapped as a cacheable memory type to a WC page, an operating system can avoid this type of aliasing by doing the following:

1. Remove the previous mapping to a cacheable memory type in the page tables; that is, make them not present.
2. Flush the TLBs of processors that may have used the mapping, even speculatively.
3. Create a new mapping to the same physical address with a new memory type, for instance, WC.
4. Flush the caches on all processors that may have used the mapping previously. Note on processors that support self-snooping, CPUID feature flag bit 27, this step is unnecessary.

Operating systems that use a page directory as a page table (to map large pages) and enable page size extensions must carefully scrutinize the use of the PAT index bit for the 4-KByte page-table entries. The PAT index bit for a page-table entry (bit 7) corresponds to the page size bit in a page-directory entry. Therefore, the operating system can only use PAT entries PA0 through PA3 when setting the caching type for a page table that is also used as a page directory. If the operating system attempts to use PAT entries PA4 through PA7 when using this memory as a page table, it effectively sets the PS bit for the access to this memory as a page directory.

For compatibility with earlier IA-32 processors that do not support the PAT, care should be taken in selecting the encodings for entries in the PAT (see Section 11.12.5, "PAT Compatibility with Earlier IA-32 Processors").

11.12.5 PAT Compatibility with Earlier IA-32 Processors

For IA-32 processors that support the PAT, the IA32_PAT MSR is always active. That is, the PCD and PWT bits in page-table entries and in page-directory entries (that point to pages) are always select a memory type for a page indirectly by selecting an entry in the PAT. They never select the memory type for a page directly as they do in earlier IA-32 processors that do not implement the PAT (see Table 11-6).

To allow compatibility for code written to run on earlier IA-32 processor that do not support the PAT, the PAT mechanism has been designed to allow backward compatibility to earlier processors. This compatibility is provided through the ordering of the PAT, PCD, and PWT bits in the 3-bit PAT entry index. For processors that do not implement the PAT, the PAT index bit (bit 7 in the page-table entries and bit 12 in the page-directory entries) is reserved and set to 0. With the PAT bit reserved, only the first four entries of the PAT can be selected with the PCD and PWT bits. At power-up or reset (see Table 11-12), these first four entries are encoded to select the same memory types as the PCD and PWT bits would normally select directly in an IA-32 processor that does not implement the PAT. So, if encodings of the first four entries in the PAT are left unchanged following a power-up or reset, code written to run on earlier IA-32 processors that do not implement the PAT will run correctly on IA-32 processors that do implement the PAT.

21. Updates to Chapter 14, Volume 3B

Change bars and green text show changes to Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

Changes to this chapter: Added new Section 14.6 "Hardware Feedback Interface".

This chapter describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

14.1 ENHANCED INTEL SPEEDSTEP® TECHNOLOGY

Enhanced Intel SpeedStep® Technology was introduced in the Pentium M processor. The technology enables the management of processor power consumption via performance state transitions. These states are defined as discrete operating points associated with different voltages and frequencies.

Enhanced Intel SpeedStep Technology differs from previous generations of Intel SpeedStep® Technology in two ways:

- Centralization of the control mechanism and software interface in the processor by using model-specific registers.
- Reduced hardware overhead; this permits more frequent performance state transitions.

Previous generations of the Intel SpeedStep Technology require processors to be a deep sleep state, holding off bus master transfers for the duration of a performance state transition. Performance state transitions under the Enhanced Intel SpeedStep Technology are discrete transitions to a new target frequency.

Support is indicated by CPUID, using ECX feature bit 07. Enhanced Intel SpeedStep Technology is enabled by setting IA32_MISC_ENABLE MSR, bit 16. On reset, bit 16 of IA32_MISC_ENABLE MSR is cleared.

14.1.1 Software Interface For Initiating Performance State Transitions

State transitions are initiated by writing a 16-bit value to the IA32_PERF_CTL register, see Figure 14-2. If a transition is already in progress, transition to a new value will subsequently take effect.

Reads of IA32_PERF_CTL determine the last targeted operating point. The current operating point can be read from IA32_PERF_STATUS. IA32_PERF_STATUS is updated dynamically.

The 16-bit encoding that defines valid operating points is model-specific. Applications and performance tools are not expected to use either IA32_PERF_CTL or IA32_PERF_STATUS and should treat both as reserved. Performance monitoring tools can access model-specific events and report the occurrences of state transitions.

14.2 P-STATE HARDWARE COORDINATION

The Advanced Configuration and Power Interface (ACPI) defines performance states (P-states) that are used to facilitate system software's ability to manage processor power consumption. Different P-states correspond to different performance levels that are applied while the processor is actively executing instructions. Enhanced Intel SpeedStep Technology supports P-states by providing software interfaces that control the operating frequency and voltage of a processor.

With multiple processor cores residing in the same physical package, hardware dependencies may exist for a subset of logical processors on a platform. These dependencies may impose requirements that impact the coordination of P-state transitions. As a result, multi-core processors may require an OS to provide additional software support for coordinating P-state transitions for those subsets of logical processors.

ACPI firmware can choose to expose P-states as dependent and hardware-coordinated to OS power management (OSPM) policy. To support OSPMs, multi-core processors must have additional built-in support for P-state hardware coordination and feedback.

Intel 64 and IA-32 processors with dependent P-states amongst a subset of logical processors permit hardware coordination of P-states and provide a hardware-coordination feedback mechanism using IA32_MPERF MSR and

IA32_APERF MSR. See Figure 14-1 for an overview of the two 64-bit MSRs and the bullets below for a detailed description.

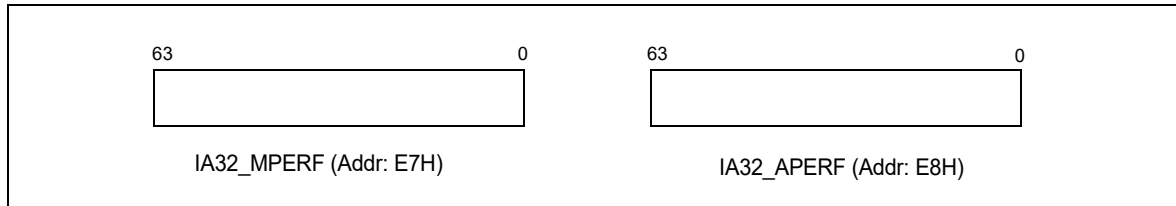


Figure 14-1. IA32_MPERF MSR and IA32_APERF MSR for P-state Coordination

- Use CPUID to check the P-State hardware coordination feedback capability bit. CPUID.06H.ECX[Bit 0] = 1 indicates IA32_MPERF MSR and IA32_APERF MSR are present.
- IA32_MPERF MSR (E7H) increments in proportion to a fixed frequency, which is configured when the processor is booted.
- IA32_APERF MSR (E8H) increments in proportion to actual performance, while accounting for hardware coordination of P-state and TM1/TM2; or software initiated throttling.
- The MSRs are per logical processor; they measure performance only when the targeted processor is in the C0 state.
- Only the IA32_APERF/IA32_MPERF ratio is architecturally defined; software should not attach meaning to the content of the individual of IA32_APERF or IA32_MPERF MSRs.
- When either MSR overflows, both MSRs are reset to zero and continue to increment.
- Both MSRs are full 64-bits counters. Each MSR can be written to independently. However, software should follow the guidelines illustrated in Example 14-1.

If P-states are exposed by the BIOS as hardware coordinated, software is expected to confirm processor support for P-state hardware coordination feedback and use the feedback mechanism to make P-state decisions. The OSPM is expected to either save away the current MSR values (for determination of the delta of the counter ratio at a later time) or reset both MSRs (execute WRMSR with 0 to these MSRs individually) at the start of the time window used for making the P-state decision. When not resetting the values, overflow of the MSRs can be detected by checking whether the new values read are less than the previously saved values.

Example 14-1 demonstrates steps for using the hardware feedback mechanism provided by IA32_APERF MSR and IA32_MPERF MSR to determine a target P-state.

Example 14-1. Determine Target P-state From Hardware Coordinated Feedback

```
DWORD PercentBusy; // Percentage of processor time not idle.
// Measure "PercentBusy" during previous sampling window.
// Typically, "PercentBusy" is measure over a time scale suitable for
// power management decisions
//
// RDMSR of MCNT and ACNT should be performed without delay.
// Software needs to exercise care to avoid delays between
// the two RDMSRs (for example, interrupts).
MCNT = RDMSR(IA32_MPERF);
ACNT = RDMSR(IA32_APERF);

// PercentPerformance indicates the percentage of the processor
// that is in use. The calculation is based on the PercentBusy,
// that is the percentage of processor time not idle and the P-state
// hardware coordinated feedback using the ACNT/MCNT ratio.
// Note that both values need to be calculated over the same
```



```

// time window.
    PercentPerformance = PercentBusy * (ACNT/MCNT);

// This example does not cover the additional logic or algorithms
// necessary to coordinate multiple logical processors to a target P-state.

TargetPstate = FindPstate(PercentPerformance);

if (TargetPstate ≠ currentPstate) {
    SetPState(TargetPstate);
}
// WRMSR of MCNT and ACNT should be performed without delay.
// Software needs to exercise care to avoid delays between
// the two WRMSRs (for example, interrupts).
WRMSR(IA32_MPERF, 0);
WRMSR(IA32_APERF, 0);

```

14.3 SYSTEM SOFTWARE CONSIDERATIONS AND OPPORTUNISTIC PROCESSOR PERFORMANCE OPERATION

An Intel 64 processor may support a form of processor operation that takes advantage of design headroom to opportunistically increase performance. The Intel[®] Turbo Boost Technology can convert thermal headroom into higher performance across multi-threaded and single-threaded workloads. The Intel[®] Dynamic Acceleration Technology feature can convert thermal headroom into higher performance if only one thread is active.

14.3.1 Intel[®] Dynamic Acceleration Technology

The Intel Core 2 Duo processor T 7700 introduces Intel Dynamic Acceleration Technology. Intel Dynamic Acceleration Technology takes advantage of thermal design headroom and opportunistically allows a single core to operate at a higher performance level when the operating system requests increased performance.

14.3.2 System Software Interfaces for Opportunistic Processor Performance Operation

Opportunistic processor performance operation, applicable to Intel Dynamic Acceleration Technology and Intel[®] Turbo Boost Technology, has the following characteristics:

- A transition from a normal state of operation (e.g. Intel Dynamic Acceleration Technology/Turbo mode disengaged) to a target state is not guaranteed, but may occur opportunistically after the corresponding enable mechanism is activated, the headroom is available and certain criteria are met.
- The opportunistic processor performance operation is generally transparent to most application software.
- System software (BIOS and Operating system) must be aware of hardware support for opportunistic processor performance operation and may need to temporarily disengage opportunistic processor performance operation when it requires more predictable processor operation.
- When opportunistic processor performance operation is engaged, the OS should use hardware coordination feedback mechanisms to prevent un-intended policy effects if it is activated during inappropriate situations.

14.3.2.1 Discover Hardware Support and Enabling of Opportunistic Processor Performance Operation

If an Intel 64 processor has hardware support for opportunistic processor performance operation, the power-on default state of IA32_MISC_ENABLE[38] indicates the presence of such hardware support. For Intel 64 processors that support opportunistic processor performance operation, the default value is 1, indicating its presence. For processors that do not support opportunistic processor performance operation, the default value is 0. The power-

on default value of IA32_MISC_ENABLE[38] allows BIOS to detect the presence of hardware support of opportunistic processor performance operation.

IA32_MISC_ENABLE[38] is shared across all logical processors in a physical package. It is written by BIOS during platform initiation to enable/disable opportunistic processor performance operation in conjunction of OS power management capabilities, see Section 14.3.2.2. BIOS can set IA32_MISC_ENABLE[38] with 1 to disable opportunistic processor performance operation; it must clear the default value of IA32_MISC_ENABLE[38] to 0 to enable opportunistic processor performance operation. OS and applications must use CPUID leaf 06H if it needs to detect processors that have opportunistic processor performance operation enabled.

When CPUID is executed with EAX = 06H on input, Bit 1 of EAX in Leaf 06H (i.e. CPUID.06H:EAX[1]) indicates opportunistic processor performance operation, such as Intel Dynamic Acceleration Technology, has been enabled by BIOS.

Opportunistic processor performance operation can be disabled by setting bit 38 of IA32_MISC_ENABLE. This mechanism is intended for BIOS only. If IA32_MISC_ENABLE[38] is set, CPUID.06H:EAX[1] will return 0.

14.3.2.2 OS Control of Opportunistic Processor Performance Operation

There may be phases of software execution in which system software cannot tolerate the non-deterministic aspects of opportunistic processor performance operation. For example, when calibrating a real-time workload to make a CPU reservation request to the OS, it may be undesirable to allow the possibility of the processor delivering increased performance that cannot be sustained after the calibration phase.

System software can temporarily disengage opportunistic processor performance operation by setting bit 32 of the IA32_PERF_CTL MSR (0199H), using a read-modify-write sequence on the MSR. The opportunistic processor performance operation can be re-engaged by clearing bit 32 in IA32_PERF_CTL MSR, using a read-modify-write sequence. The DISENAGE bit in IA32_PERF_CTL is not reflected in bit 32 of the IA32_PERF_STATUS MSR (0198H), and it is not shared between logical processors in a physical package. In order for OS to engage Intel Dynamic Acceleration Technology/Turbo mode, the BIOS must:

- Enable opportunistic processor performance operation, as described in Section 14.3.2.1.
- Expose the operating points associated with Intel Dynamic Acceleration Technology/Turbo mode to the OS.

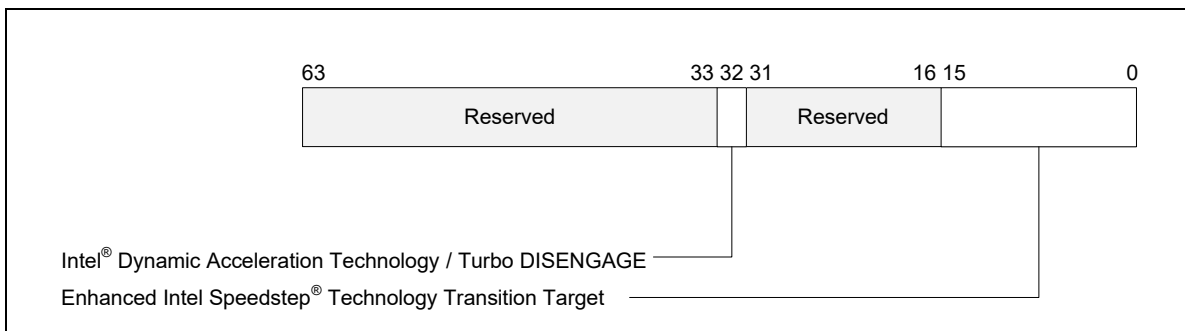


Figure 14-2. IA32_PERF_CTL Register

14.3.2.3 Required Changes to OS Power Management P-State Policy

Intel Dynamic Acceleration Technology and Intel Turbo Boost Technology can provide opportunistic performance greater than the performance level corresponding to the Processor Base frequency of the processor (see CPUID's processor frequency information). System software can use a pair of MSRs to observe performance feedback. Software must query for the presence of IA32_APERF and IA32_MPERF (see Section 14.2). The ratio between IA32_APERF and IA32_MPERF is architecturally defined and a value greater than unity indicates performance increase occurred during the observation period due to Intel Dynamic Acceleration Technology. Without incorporating such performance feedback, the target P-state evaluation algorithm can result in a non-optimal P-state target.

There are other scenarios under which OS power management may want to disable Intel Dynamic Acceleration Technology, some of these are listed below:

- When engaging ACPI defined passive thermal management, it may be more effective to disable Intel Dynamic Acceleration Technology for the duration of passive thermal management.
- When the user has indicated a policy preference of power savings over performance, OS power management may want to disable Intel Dynamic Acceleration Technology while that policy is in effect.

14.3.3 Intel® Turbo Boost Technology

Intel Turbo Boost Technology is supported in Intel Core i7 processors and Intel Xeon processors based on Intel® microarchitecture code name Nehalem. It uses the same principle of leveraging thermal headroom to dynamically increase processor performance for single-threaded and multi-threaded/multi-tasking environment. The programming interface described in Section 14.3.2 also applies to Intel Turbo Boost Technology.

14.3.4 Performance and Energy Bias Hint Support

Intel 64 processors may support additional software hint to guide the hardware heuristic of power management features to favor increasing dynamic performance or conserve energy consumption.

Software can detect the processor's capability to support the performance-energy bias preference hint by examining bit 3 of ECX in CPUID leaf 6. The processor supports this capability 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).

Software can program the lowest four bits of IA32_ENERGY_PERF_BIAS MSR with a value from 0 - 15. The values represent a sliding scale, where a value of 0 (the default reset value) corresponds to a hint preference for highest performance and a value of 15 corresponds to the maximum energy savings. A value of 7 roughly translates into a hint to balance performance with energy consumption.

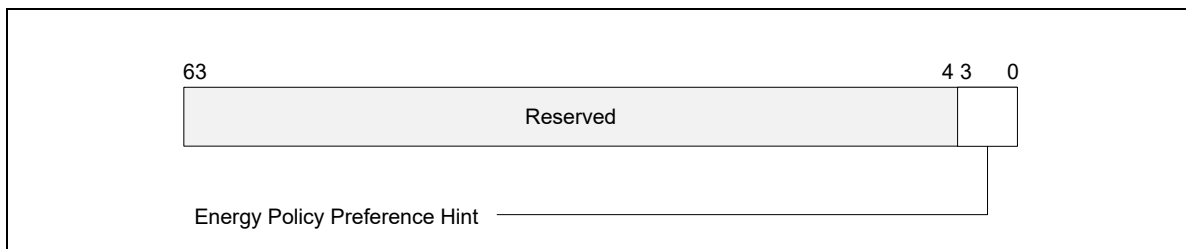


Figure 14-3. IA32_ENERGY_PERF_BIAS Register

The layout of IA32_ENERGY_PERF_BIAS is shown in Figure 14-3. The scope of IA32_ENERGY_PERF_BIAS is per logical processor, which means that each of the logical processors in the package can be programmed with a different value. This may be especially important in virtualization scenarios, where the performance / energy requirements of one logical processor may differ from the other. Conflicting "hints" from various logical processors at higher hierarchy level will be resolved in favor of performance over energy savings.

Software can use whatever criteria it sees fit to program the MSR with an appropriate value. However, the value only serves as a hint to the hardware and the actual impact on performance and energy savings is model specific.

14.4 HARDWARE-CONTROLLED PERFORMANCE STATES (HWP)

Intel processors may contain support for Hardware-Controlled Performance States (HWP), which autonomously selects performance states while utilizing OS supplied performance guidance hints. The Enhanced Intel Speed-Step® Technology provides a means for the OS to control and monitor discrete frequency-based operating points via the IA32_PERF_CTL and IA32_PERF_STATUS MSRs.

In contrast, HWP is an implementation of the ACPI-defined Collaborative Processor Performance Control (CPPC), which specifies that the platform enumerates a continuous, abstract unit-less, performance value scale that is not tied to a specific performance state / frequency by definition. While the enumerated scale is roughly linear in terms of a delivered integer workload performance result, the OS is required to characterize the performance value range to comprehend the delivered performance for an applied workload.

When HWP is enabled, the processor autonomously selects performance states as deemed appropriate for the applied workload and with consideration of constraining hints that are programmed by the OS. These OS-provided hints include minimum and maximum performance limits, preference towards energy efficiency or performance, and the specification of a relevant workload history observation time window. The means for the OS to override HWP's autonomous selection of performance state with a specific desired performance target is also provided, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations.

14.4.1 HWP Programming Interfaces

The programming interfaces provided by HWP include the following:

- The CPUID instruction allows software to discover the presence of HWP support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input will return 5 bit flags covering the following aspects in bits 7 through 11 of CPUID.06H:EAX:
 - Availability of HWP baseline resource and capability, CPUID.06H:EAX[bit 7]: If this bit is set, HWP provides several new architectural MSRs: IA32_PM_ENABLE, IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS.
 - Availability of HWP Notification upon dynamic Guaranteed Performance change, CPUID.06H:EAX[bit 8]: If this bit is set, HWP provides IA32_HWP_INTERRUPT MSR to enable interrupt generation due to dynamic Performance changes and excursions.
 - Availability of HWP Activity window control, CPUID.06H:EAX[bit 9]: If this bit is set, HWP allows software to program activity window in the IA32_HWP_REQUEST MSR.
 - Availability of HWP energy/performance preference control, CPUID.06H:EAX[bit 10]: If this bit is set, HWP allows software to set an energy/performance preference hint in the IA32_HWP_REQUEST MSR.
 - Availability of HWP package level control, CPUID.06H:EAX[bit 11]: If this bit is set, HWP provides the IA32_HWP_REQUEST_PKG MSR to convey OS Power Management's control hints for all logical processors in the physical package.

Table 14-1. Architectural and Non-Architectural MSRs Related to HWP

Address	Architectural	Register Name	Description
770H	Y	IA32_PM_ENABLE	Enable/Disable HWP.
771H	Y	IA32_HWP_CAPABILITIES	Enumerates the HWP performance range (static and dynamic).
772H	Y	IA32_HWP_REQUEST_PKG	Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for all logical processor in the physical package.
773H	Y	IA32_HWP_INTERRUPT	Controls HWP native interrupt generation (Guaranteed Performance changes, excursions).
774H	Y	IA32_HWP_REQUEST	Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for a single logical processor.
775H	Y	IA32_HWP_PECI_REQUEST_INFO	Conveys embedded system controller requests to override some of the OS HWP Request settings via the PECI mechanism.
777H	Y	IA32_HWP_STATUS	Status bits indicating changes to Guaranteed Performance and excursions to Minimum Performance.
19CH	Y	IA32_THERM_STATUS[bits 15:12]	Conveys reasons for performance excursions.
64EH	N	MSR_PPERF	Productive Performance Count.

- Additionally, HWP may provide a non-architectural MSR, MSR_PPERF, which provides a quantitative metric to software of hardware's view of workload scalability. This hardware's view of workload scalability is implementation specific.

14.4.2 Enabling HWP

The layout of the IA32_PM_ENABLE MSR is shown in Figure 14-4. The bit fields are described below:

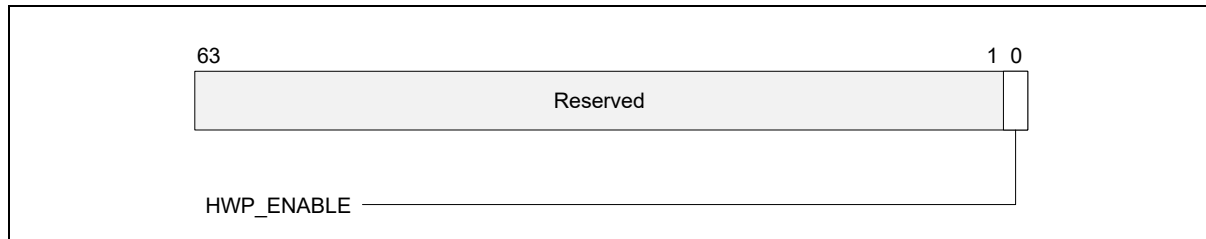


Figure 14-4. IA32_PM_ENABLE MSR

- **HWP_ENABLE (bit 0, R/W1Once)** — Software sets this bit to enable HWP with autonomous selection of processor P-States. When set, the processor will disregard input from the legacy performance control interface (IA32_PERF_CTL). Note this bit can only be enabled once from the default value. Once set, writes to the HWP_ENABLE bit are ignored. Only RESET will clear this bit. Default = zero (0).
- Bits 63:1 are reserved and must be zero.

After software queries CPUID and verifies the processor's support of HWP, system software can write 1 to IA32_PM_ENABLE.HWP_ENABLE (bit 0) to enable hardware controlled performance states. The default value of IA32_PM_ENABLE MSR at power-on is 0, i.e. HWP is disabled.

Additional MSRs associated with HWP may only be accessed after HWP is enabled, with the exception of IA32_HWP_INTERRUPT and MSR_PPERF. Accessing the IA32_HWP_INTERRUPT MSR requires only HWP is present as enumerated by CPUID but does not require enabling HWP.

IA32_PM_ENABLE is a package level MSR, i.e., writing to it from any logical processor within a package affects all logical processors within that package.

14.4.3 HWP Performance Range and Dynamic Capabilities

The OS reads the IA32_HWP_CAPABILITIES MSR to comprehend the limits of the HWP-managed performance range as well as the dynamic capability, which may change during processor operation. The enumerated performance range values reported by IA32_HWP_CAPABILITIES directly map to initial frequency targets (prior to workload-specific frequency optimizations of HWP). However the mapping is processor family specific.

The layout of the IA32_HWP_CAPABILITIES MSR is shown in Figure 14-5. The bit fields are described below:

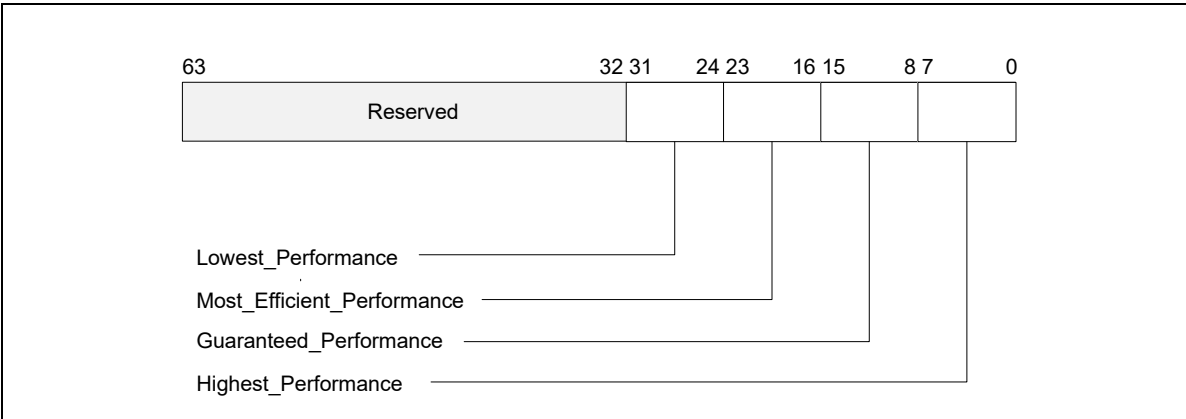


Figure 14-5. IA32_HWP_CAPABILITIES Register

- **Highest_Performance (bits 7:0, RO)** — Value for the maximum non-guaranteed performance level.
- **Guaranteed_Performance (bits 15:8, RO)** — Current value for the guaranteed performance level. This value can change dynamically as a result of internal or external constraints, e.g. thermal or power limits.
- **Most_Efficient_Performance (bits 23:16, RO)** — Current value of the most efficient performance level. This value can change dynamically as a result of workload characteristics.
- **Lowest_Performance (bits 31:24, RO)** — Value for the lowest performance level that software can program to IA32_HWP_REQUEST.
- Bits 63:32 are reserved and must be zero.

The value returned in the **Guaranteed_Performance** field is hardware's best-effort approximation of the available performance given current operating constraints. Changes to the Guaranteed_Performance value will primarily occur due to a shift in operational mode. This includes a power or other limit applied by an external agent, e.g. RAPL (see Figure 14.10.1), or the setting of a Configurable TDP level (see model-specific controls related to Programmable TDP Limit in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.*). Notification of a change to the Guaranteed_Performance occurs via interrupt (if configured) and the IA32_HWP_Status MSR. Changes to Guaranteed_Performance are indicated when a macroscopically meaningful change in performance occurs i.e. sustained for greater than one second. Consequently, notification of a change in Guaranteed Performance will typically occur no more frequently than once per second. Rapid changes in platform configuration, e.g. docking / undocking, with corresponding changes to a Configurable TDP level could potentially cause more frequent notifications.

The value returned by the **Most_Efficient_Performance** field provides the OS with an indication of the practical lower limit for the IA32_HWP_REQUEST. The processor may not honor IA32_HWP_REQUEST.Maximum Performance settings below this value.

14.4.4 Managing HWP

14.4.4.1 IA32_HWP_REQUEST MSR (Address: 0x774 Logical Processor Scope)

Typically, the operating system controls HWP operation for each logical processor via the writing of control hints / constraints to the IA32_HWP_REQUEST MSR. The layout of the IA32_HWP_REQUEST MSR is shown in Figure 14-6. The bit fields are described below Figure 14-6.

Operating systems can control HWP by writing both IA32_HWP_REQUEST and IA32_HWP_REQUEST_PKG MSRs (see Section 14.4.4.2). Five valid bits within the IA32_HWP_REQUEST MSR let the operating system flexibly select which of its five hint / constraint fields should be derived by the processor from the IA32_HWP_REQUEST MSR and which should be derived from the IA32_HWP_REQUEST_PKG MSR. These five valid bits are supported if CPUID[6].EAX[17] is set.

When the IA32_HWP_REQUEST MSR Package Control bit is set, any valid bit that is NOT set indicates to the processor to use the respective field value from the IA32_HWP_REQUEST_PKG MSR. Otherwise, the values are derived from the IA32_HWP_REQUEST MSR. The valid bits are ignored when the IA32_HWP_REQUEST MSR Package Control bit is zero.

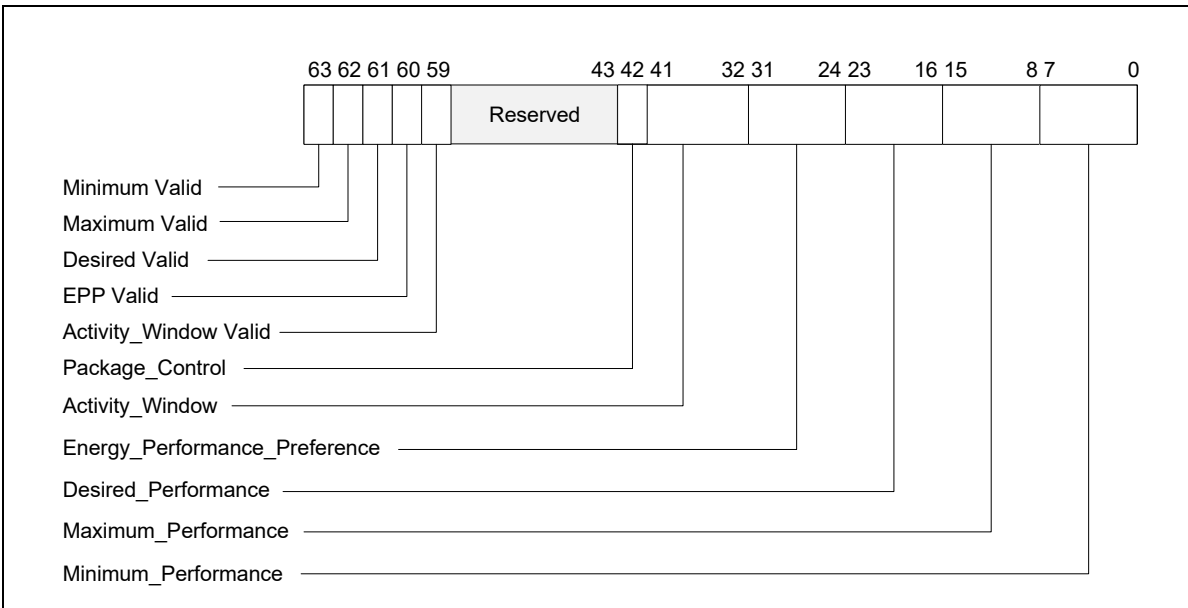


Figure 14-6. IA32_HWP_REQUEST Register

- **Minimum_Performance (bits 7:0, RW)** — Conveys a hint to the HWP hardware. The OS programs the minimum performance hint to achieve the required quality of service (QoS) or to meet a service level agreement (SLA) as needed. Note that an excursion below the level specified is possible due to hardware constraints. The default value of this field is IA32_HWP_CAPABILITIES.Lowest_Performance.
- **Maximum_Performance (bits 15:8, RW)** — Conveys a hint to the HWP hardware. The OS programs this field to limit the maximum performance that is expected to be supplied by the HWP hardware. Excursions above the limit requested by OS are possible due to hardware coordination between the processor cores and other components in the package. The default value of this field is IA32_HWP_CAPABILITIES.Highest_Performance.
- **Desired_Performance (bits 23:16, RW)** — Conveys a hint to the HWP hardware. When set to zero, hardware autonomous selection determines the performance target. When set to a non-zero value (between the range of Lowest_Performance and Highest_Performance of IA32_HWP_CAPABILITIES) conveys an explicit performance request hint to the hardware; effectively disabling HW Autonomous selection. The Desired_Performance input is non-constraining in terms of Performance and Energy Efficiency optimizations, which are independently controlled. The default value of this field is 0.
- **Energy_Performance_Preference (bits 31:24, RW)** — Conveys a hint to the HWP hardware. The OS may write a range of values from 0 (performance preference) to 0FFH (energy efficiency preference) to influence the rate of performance increase /decrease and the result of the hardware's energy efficiency and performance optimizations. The default value of this field is 80H. Note: If CPUID.06H:EAX[bit 10] indicates that this field is not supported, HWP uses the value of the IA32_ENERGY_PERF_BIAS MSR to determine the energy efficiency / performance preference.
- **Activity_Window (bits 41:32, RW)** — Conveys a hint to the HWP hardware specifying a moving workload history observation window for performance/frequency optimizations. If 0, the hardware will determine the appropriate window size. When writing a non-zero value to this field, this field is encoded in the format of bits 38:32 as a 7-bit mantissa and bits 41:39 as a 3-bit exponent value in powers of 10. The resultant value is in microseconds. Thus, the minimal/maximum activity window size is 1 microsecond/1270 seconds. Combined with the Energy_Performance_Preference input, Activity_Window influences the rate of performance increase

/ decrease. This non-zero hint only has meaning when Desired_Performance = 0. The default value of this field is 0.

- **Package_Control (bit 42, RW)** — When set, causes this logical processor's IA32_HWP_REQUEST control inputs to be derived from the IA32_HWP_REQUEST_PKG MSR.
- Bits 58:43 are reserved and must be zero.
- **Activity_Window_Valid (bit 59, RW)** — When set, indicates to the processor to derive the Activity Window field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **EPP_Valid (bit 60, RW)** — When set, indicates to the processor to derive the EPP field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Desired_Valid (bit 61, RW)** — When set, indicates to the processor to derive the Desired Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Maximum_Valid (bit 62, RW)** — When set, indicates to the processor to derive the Maximum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Minimum_Valid (bit 63, RW)** — When set, indicates to the processor to derive the Minimum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

The HWP hardware clips and resolves the field values as necessary to the valid range. Reads return the last value written not the clipped values.

Processors may support a subset of IA32_HWP_REQUEST fields as indicated by CPUID. Reads of non-supported fields will return 0. Writes to non-supported fields are ignored.

The OS may override HWP's autonomous selection of performance state with a specific performance target by setting the Desired_Performance field to a non-zero value, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations, which are influenced by the Energy Performance Preference field.

Software may disable all hardware optimizations by setting Minimum_Performance = Maximum_Performance (subject to package coordination).

Note: The processor may run below the Minimum_Performance level due to hardware constraints including: power, thermal, and package coordination constraints. The processor may also run below the Minimum_Performance level for short durations (few milliseconds) following C-state exit, and when Hardware Duty Cycling (see Section 14.5) is enabled.

When the IA32_HWP_REQUEST MSR is set to fast access mode, writes of this MSR are posted, i.e., the WRMSR instruction retires before the data reaches its destination within the processor. It may retire even before all preceding IA stores are globally visible, i.e., it is not an architecturally serializing instruction anymore (no store fence). A new CPUID bit indicates this new characteristic of the IA32_HWP_REQUEST MSR (see Section 14.4.8 for additional details).

14.4.4.2 IA32_HWP_REQUEST_PKG MSR (Address: 772H Package Scope)

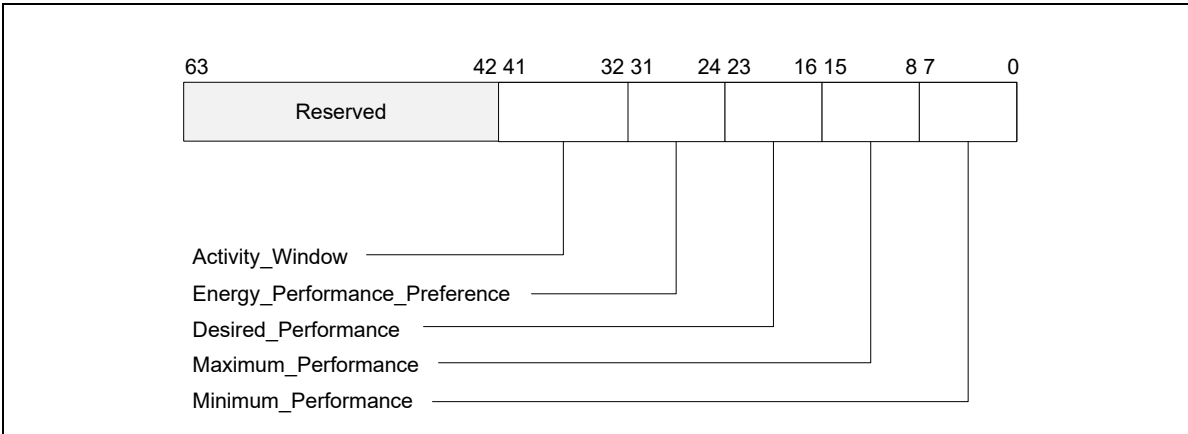


Figure 14-7. IA32_HWP_REQUEST_PKG Register

The structure of the IA32_HWP_REQUEST_PKG MSR (package-level) is identical to the IA32_HWP_REQUEST MSR with the exception of the the Package Control bit field and the five valid bit fields, which do not exist in the IA32_HWP_REQUEST_PKG MSR. Field values written to this MSR apply to all logical processors within the physical package with the exception of logical processors whose IA32_HWP_REQUEST.Package Control field is clear (zero). Single P-state Control mode is only supported when IA32_HWP_REQUEST_PKG is not supported.

14.4.4.3 IA32_HWP_PECI_REQUEST_INFO MSR (Address 775H Package Scope)

When an embedded system controller is integrated in the platform, it can override some of the OS HWP Request settings via the PECI mechanism. PECI initiated settings take precedence over the relevant fields in the IA32_HWP_REQUEST MSR and in the IA32_HWP_REQUEST_PKG MSR, irrespective of the Package Control bit or the Valid Bit values described above. PECI can independently control each of: Minimum Performance, Maximum Performance and EPP fields. This MSR contains both the PECI induced values and the control bits that indicate whether the embedded controller actually set the processor to use the respective value.

PECI override is supported if CPUID[6].EAX[16] is set.

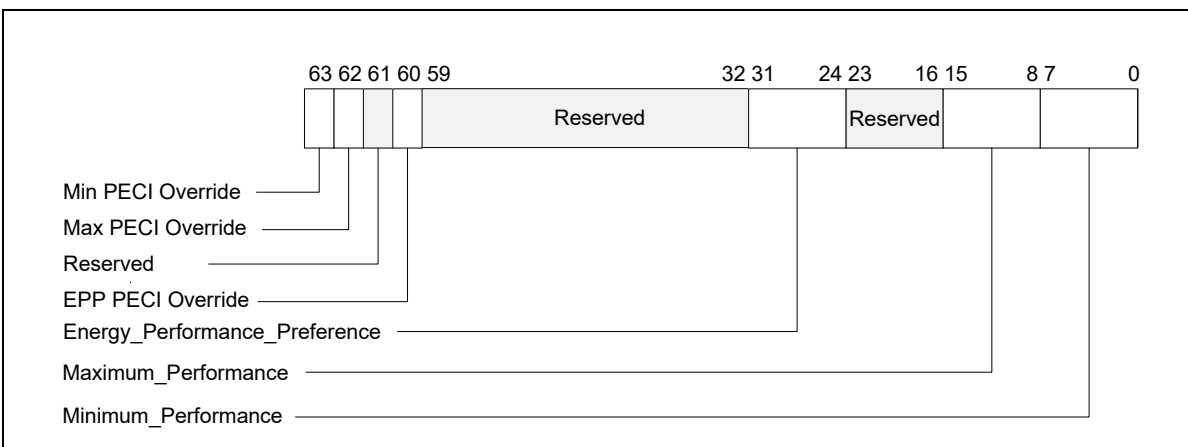


Figure 14-8. IA32_HWP_PECI_REQUEST_INFO MSR

The layout of the IA32_HWP_PECI_REQUEST_INFO MSR is shown in Figure 14-8. This MSR is writable by the embedded controller but is read-only by software executing on the CPU. This MSR has Package scope. The bit fields are described below:

- **Minimum_Performance (bits 7:0, RO)** — Used by the OS to read the latest value of Peci minimum performance input.
- **Maximum_Performance (bits 15:8, RO)** — Used by the OS to read the latest value of Peci maximum performance input.
- Bits 23:16 are reserved and must be zero.
- **Energy_Performance_Preference (bits 31:24, RO)** — Used by the OS to read the latest value of Peci energy performance preference input.
- Bits 59:32 are reserved and must be zero.
- **EPP_Peci_Override (bit 60, RO)** — Indicates whether Peci is currently overriding the Energy Performance Preference input. If set(1), Peci is overriding the Energy Performance Preference input. If clear(0), OS has control over Energy Performance Preference input.
- Bit 61 is reserved and must be zero.
- **Max_Peci_Override (bit 62, RO)** — Indicates whether Peci is currently overriding the Maximum Performance input. If set(1), Peci is overriding the Maximum Performance input. If clear(0), OS has control over Maximum Performance input.
- **Min_Peci_Override (bit 63, RO)** — Indicates whether Peci is currently overriding the Minimum Performance input. If set(1), Peci is overriding the Minimum Performance input. If clear(0), OS has control over Minimum Performance input.

HWP Request Field Hierarchical Resolution

HWP Request field resolution is fed by three MSRs: IA32_HWP_REQUEST, IA32_HWP_REQUEST_PKG and IA32_HWP_PECI_REQUEST_INFO. The flow that the processor goes through to resolve which field value is chosen is shown below.

For each of the two HWP Request fields; Desired and Activity Window:

```
If IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
    Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>
Else
    Resolved Field Value = IA32_HWP_REQUEST.<field>
```

For each of the three HWP Request fields; Min, Max and EPP:

```
If IA32_HWP_PECI_REQUEST_INFO.<field> Peci Override bit = 1
    Resolved Field Value = IA32_HWP_PECI_REQUEST_INFO.<field>
Else if IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
    Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>
Else
    Resolved Field Value = IA32_HWP_REQUEST.<field>
```

14.4.5 HWP Feedback

The processor provides several types of feedback to the OS during HWP operation.

The IA32_MPERF MSR and IA32_APERF MSR mechanism (see Section 14.2) allows the OS to calculate the resultant effective frequency delivered over a time period. Energy efficiency and performance optimizations directly impact the resultant effective frequency delivered.

The layout of the IA32_HWP_STATUS MSR is shown in Figure 14-9. It provides feedback regarding changes to IA32_HWP_CAPABILITIES.Guaranteed_Performance, IA32_HWP_CAPABILITIES.Highest_Performance, excursions to IA32_HWP_CAPABILITIES.Minimum_Performance, and Peci_Override entry/exit events. The bit fields are described below:

- **Guaranteed_Performance_Change (bit 0, RWC0)** — If set (1), a change to Guaranteed_Performance has occurred. Software should query IA32_HWP_CAPABILITIES.Guaranteed_Performance value to ascertain the new Guaranteed Performance value and to assess whether to re-adjust HWP hints via IA32_HWP_REQUEST. Software must clear this bit by writing a zero (0).
- Bit 1 is reserved and must be zero.
- **Excursion_To_Minimum (bit 2, RWC0)** — If set (1), an excursion to Minimum_Performance of IA32_HWP_REQUEST has occurred. Software must clear this bit by writing a zero (0).
- **Highest_Change (bit 3, RWC0)** — If set (1), a change to Highest Performance has occurred. Software should query IA32_HWP_CAPABILITIES to ascertain the new Highest Performance value. Software must clear this bit by writing a zero (0). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.
- **PECI_Override_Entry (bit 4, RWC0)** — If set (1), an embedded/management controller has started a PECI override of one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are now overridden via the PECI mechanism and what their values are (see Section 14.4.4.3 for additional details). Software must clear this bit by writing a zero (0). Interrupts upon PECI override entry are supported if CPUID[6].EAX[16] is set.
- **PECI_Override_Exit (bit 5, RWC0)** — If set (1), an embedded/management controller has stopped overriding one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are still overridden via the PECI mechanism and which fields are now back under software control (see Section 14.4.4.3 for additional details). Software must clear this bit by writing a zero (0). Interrupts upon PECI override exit are supported if CPUID[6].EAX[16] is set.
- Bits 63:6 are reserved and must be zero.

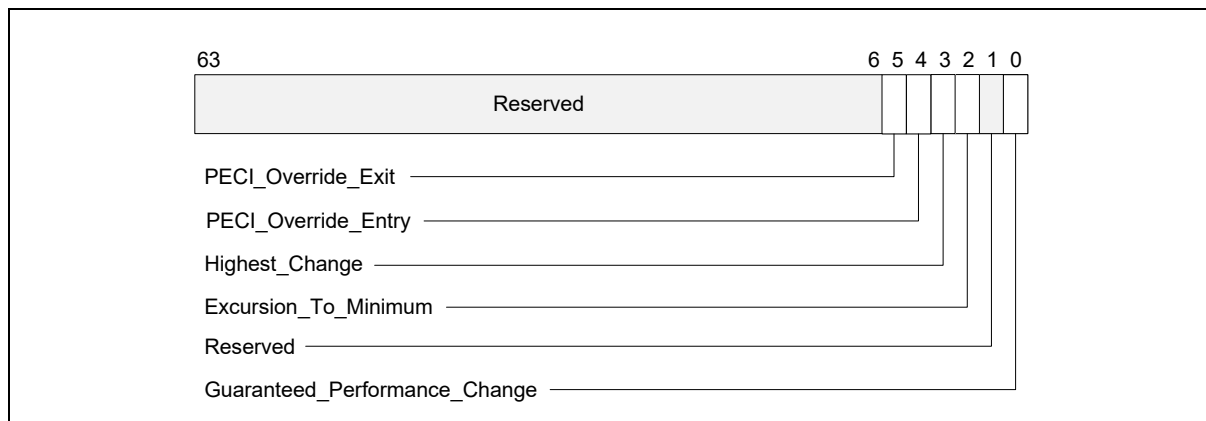


Figure 14-9. IA32_HWP_STATUS MSR

The status bits of IA32_HWP_STATUS must be cleared (0) by software so that a new status condition change will cause the hardware to set the bit again and issue the notification. Status bits are not set for “normal” excursions, e.g., running below Minimum Performance for short durations during C-state exit. Changes to Guaranteed_Performance, Highest_Performance, excursions to Minimum_Performance, or PECI_Override entry/exit will occur no more than once per second.

The OS can determine the specific reasons for a Guaranteed_Performance change or an excursion to Minimum_Performance in IA32_HWP_REQUEST by examining the associated status and log bits reported in the IA32_THERM_STATUS MSR. The layout of the IA32_HWP_STATUS MSR that HWP uses to support software query of HWP feedback is shown in Figure 14-10. The bit fields of IA32_THERM_STATUS associated with HWP feedback are described below (Bit fields of IA32_THERM_STATUS unrelated to HWP can be found in Section 14.8.5.2).

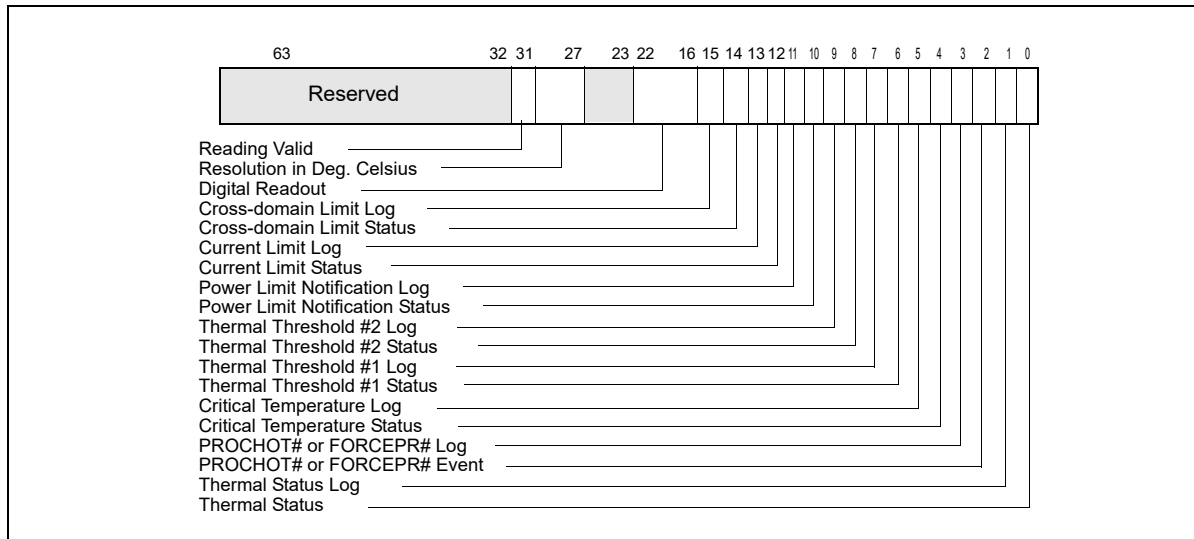


Figure 14-10. IA32_THERM_STATUS Register With HWP Feedback

- Bits 11:0, See Section 14.8.5.2.
- **Current Limit Status (bit 12, RO)** — If set (1), indicates an electrical current limit (e.g. Electrical Design Point/IccMax) is being exceeded and is adversely impacting energy efficiency optimizations.
- **Current Limit Log (bit 13, RWC0)** — If set (1), an electrical current limit has been exceeded that has adversely impacted energy efficiency optimizations since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).
- **Cross-domain Limit Status (bit 14, RO)** — If set (1), indicates another hardware domain (e.g. processor graphics) is currently limiting energy efficiency optimizations in the processor core domain.
- **Cross-domain Limit Log (bit 15, RWC0)** — If set (1), indicates another hardware domain (e.g. processor graphics) has limited energy efficiency optimizations in the processor core domain since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).
- Bits 63:16, See Section 14.8.5.2.

14.4.5.1 Non-Architectural HWP Feedback

The Productive Performance (MSR_PPERR) MSR (non-architectural) provides hardware's view of workload scalability, which is a rough assessment of the relationship between frequency and workload performance, to software. The layout of the MSR_PPERR is shown in Figure 14-11.

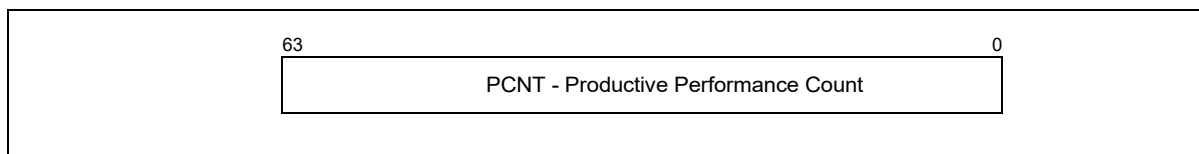


Figure 14-11. MSR_PPERR MSR

- **PCNT (bits 63:0, RO)** — Similar to IA32_APERF but only counts cycles perceived by hardware as contributing to instruction execution (e.g. unhalted and unstalled cycles). This counter increments at the same rate as IA32_APERF, where the ratio of ($\Delta PCNT / \Delta ACNT$) is an indicator of workload scalability (0% to 100%). Note that values in this register are valid even when HWP is not enabled.

14.4.6 HWP Notifications

Processors may support interrupt-based notification of changes to HWP status as indicated by CPUID. If supported, the IA32_HWP_INTERRUPT MSR is used to enable interrupt-based notifications. Notification events, when enabled, are delivered using the existing thermal LVT entry. The layout of the IA32_HWP_INTERRUPT is shown in Figure 14-12. The bit fields are described below:

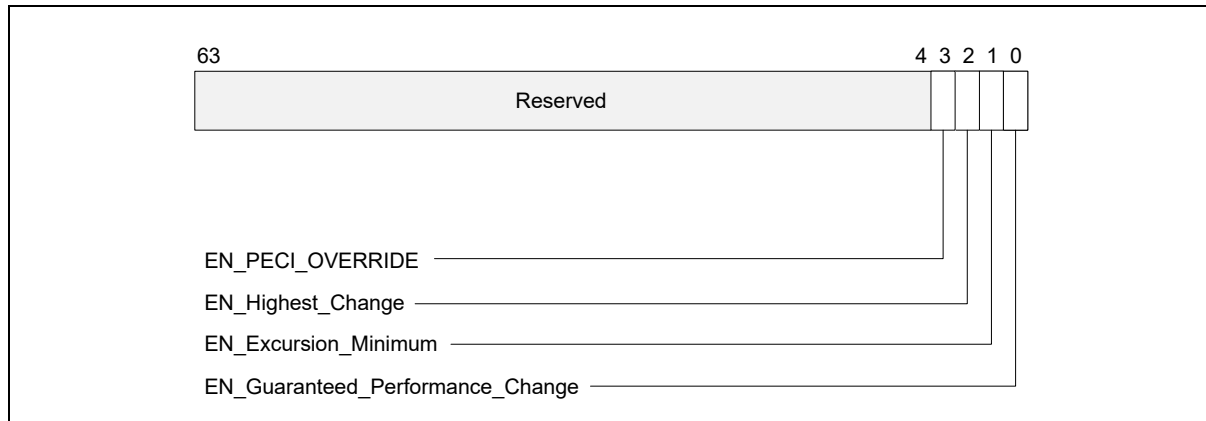


Figure 14-12. IA32_HWP_INTERRUPT MSR

- **EN_Guaranteed_Performance_Change (bit 0, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Guaranteed_Performance occurs. The default value is 0 (Interrupt generation is disabled).
- **EN_Excursion_Minimum (bit 1, RW)** — When set (1), an HWP Interrupt will be generated whenever the HWP hardware is unable to meet the IA32_HWP_REQUEST.Minimum_Performance setting. The default value is 0 (Interrupt generation is disabled).
- **EN_Highest_Change (bit 2, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Highest_Performance occurs. The default value is 0 (interrupt generation is disabled). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.
- **EN_PECI_OVERRIDE (bit 3, RW)** — When set (1), an HWP Interrupt will be generated whenever PECI starts or stops overriding any of the three HWP fields described in Section 14.4.4.3. The default value is 0 (interrupt generation is disabled). See Section 14.4.5 and Section 14.4.4.3 for details on how the OS learns what is the current set of HWP fields that are overridden by PECI. Interrupts upon PECI override change are supported if CPUID[6].EAX[16] is set.
- Bits 63:4 are reserved and must be zero.

14.4.7 Idle Logical Processor Impact on Core Frequency

Intel processors use one of two schemes for setting core frequency:

1. All cores share same frequency.
2. Each physical core is set to a frequency of its own.

In both cases the two logical processors that share a single physical core are set to the same frequency, so the processor accounts for the IA32_HWP_REQUEST MSR fields of both logical processors when defining the core frequency or the whole package frequency.

When **CPUID[6].EAX[20]** is set and only one logical processor of the two is active, while the other is idle (in any **C1 sub-state** or in a deeper sleep state), only the **active logical processor's** IA32_HWP_REQUEST MSR fields are considered, i.e., the HWP Request fields of a logical processor in the C1E sub-state or in a deeper sleep state are ignored.

Note: when a logical processor is in **C1 state** its HWP Request fields are accounted for.

14.4.8 Fast Write of Uncore MSR (Model Specific Feature)

There are a few logical processor scope MSRs whose values need to be observed outside the logical processor. The WRMSR instruction takes over 1000 cycles to complete (retire) for those MSRs. This overhead forces operating systems to avoid writing them too often whereas in many cases it is preferable that the OS writes them quite frequently for optimal power/performance operation of the processor.

The model specific “Fast Write MSR” feature reduces this overhead by an order of magnitude to a level of 100 cycles for a selected subset of MSRs.

Note: Writes to Fast Write MSRs are posted, i.e., when the WRMSR instruction completes, the data may still be “in transit” within the processor. Software can check the status by querying the processor to ensure data is already visible outside the logical processor (see Section 14.4.8.3 for additional details). Once the data is visible outside the logical processor, software is ensured that later writes by the same logical processor to the same MSR will be visible later (will not bypass the earlier writes).

MSRs that are selected for Fast Write are specified in a special capability MSR (see Section 14.4.8.1). Architectural MSRs that existed prior to the introduction of this feature and are selected for Fast Write, thus turning from slow to fast write MSRs, will be noted as such via a new CPUID bit. New MSRs that are fast upon introduction will be documented as such without an additional CPUID bit.

Three model specific MSRs are associated with the feature itself. They enable *enumerating, controlling and monitoring* it. All three are logical processor scope.

14.4.8.1 FAST_UNCORE_MSRS_CAPABILITY (Address: 0x65F, Logical Processor Scope)

Operating systems or BIOS can read the FAST_UNCORE_MSRS_CAPABILITY MSR to enumerate those MSRs that are Fast Write MSRs.

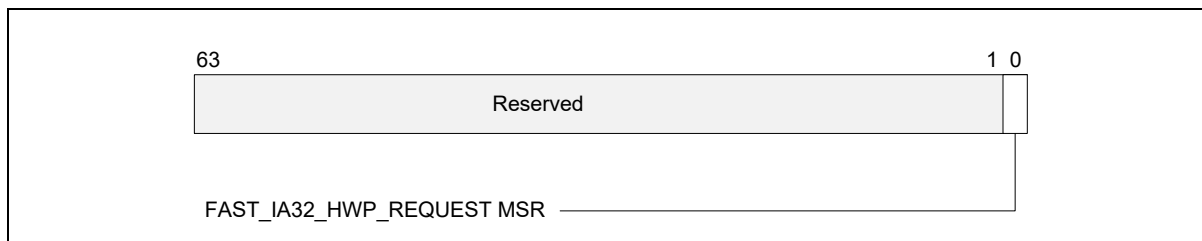


Figure 14-13. FAST_UNCORE_MSRS_CAPABILITY MSR

- **FAST_IA32_HWP_REQUEST MSR (bit 0, RO)** — When set (1), indicates that the IA32_HWP_REQUEST MSR is supported as a Fast Write MSR. A value of 0 indicates the IA32_HWP_REQUEST MSR is not supported as a Fast Write MSR.
- Bits 63:1 are reserved and must be zero.

14.4.8.2 FAST_UNCORE_MSRS_CTL (Address: 0x657, Logical Processor Scope)

Operating Systems or BIOS can use the FAST_UNCORE_MSRS_CTL MSR to opt-in or opt-out for fast write of specific MSRs that are enabled for Fast Write by the processor.

Note: Not all MSRs that are selected for this feature will necessarily have this opt-in/opt-out option. They may be supported in fast write mode only.

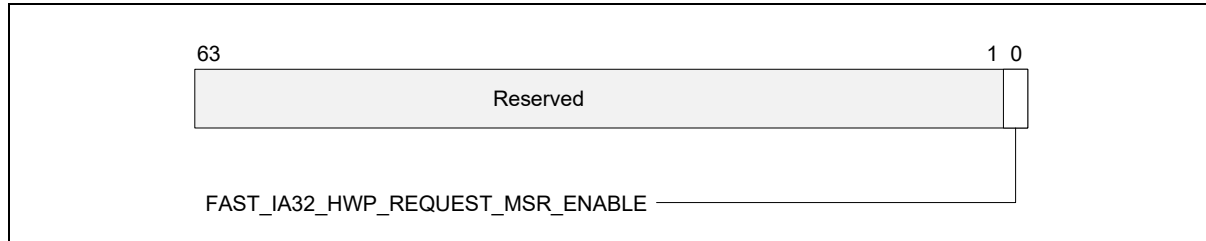


Figure 14-14. FAST_UNCORE_MSRS_CTL MSR

- **FAST_IA32_HWP_REQUEST_MSR_ENABLE (bit 0, RW)** — When set (1), enables fast access mode for the IA32_HWP_REQUEST MSR and sets the low latency, posted IA32_HWP_REQUEST MSR' CPUID[6].EAX[18]. The default value is 0. Note that this bit can only be enabled once from the default value. Once set, writes to this bit are ignored. Only RESET will clear this bit.
- Bits 63:1 are reserved and must be zero.

14.4.8.3 FAST_UNCORE_MSRS_STATUS (Address: 0x65E, Logical Processor Scope)

Software that executes the WRMSR instruction of a Fast Write MSR can check whether the data is already visible outside the logical processor by reading the FAST_UNCORE_MSRS_STATUS MSR. For each Fast Write MSR there is a status bit that indicates whether the data is already visible outside the logical processor or is still in "transit".

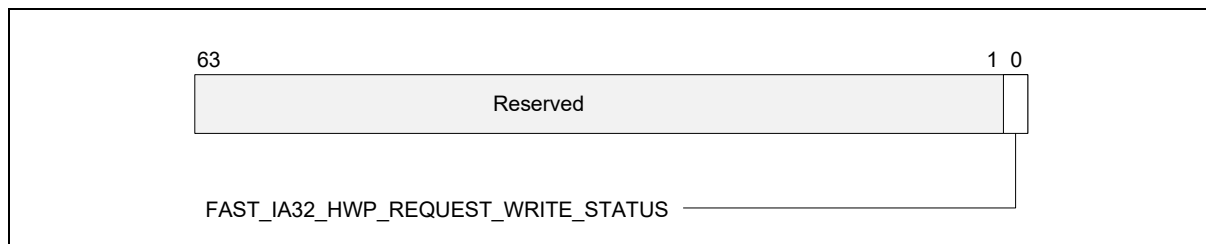


Figure 14-15. FAST_UNCORE_MSRS_STATUS MSR

- **FAST_IA32_HWP_REQUEST_WRITE_STATUS (bit 0, RO)** — Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor.
- Bits 63:1 are reserved and must be zero.

14.4.9 Fast_IA32_HWP_REQUEST CPUID

IA32_HWP_REQUEST is an architectural MSR that exists in processors whose CPUID[6].EAX[7] is set (HWP BASE is enabled). This MSR has logical processor scope, but after its contents are written the contents become visible outside the logical processor. When the FAST_IA32_HWP_REQUEST CPUID[6].EAX[18] bit is set, writes to the IA32_HWP_REQUEST MSR are visible outside the logical processor via the "Fast Write" feature described in Section 14.4.8.

14.4.10 Recommendations for OS use of HWP Controls

Common Cases of Using HWP

The default HWP control field values are expected to be suitable for many applications. The OS can enable autonomous HWP for these common cases by

- Setting IA32_HWP_REQUEST.Desired_Performance = 0 (hardware autonomous selection determines the performance target). Set IA32_HWP_REQUEST.Activity_Window = 0 (enable HW dynamic selection of window size).

To maximize HWP benefit for the common cases, the OS should set

- IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and
- IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance.

Setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance is functionally equivalent to using of the IA32_PERF_CTL interface and is therefore not recommended (bypassing HWP).

Calibrating HWP for Application-Specific HWP Optimization

In some applications, the OS may have Quality of Service requirements that may not be met by the default values. The OS can characterize HWP by:

- keeping IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance to prevent non-linearity in the characterization process,
- utilizing the range values enumerated from the IA32_HWP_CAPABILITIES MSR to program IA32_HWP_REQUEST while executing workloads of interest and observing the power and performance result.

The power and performance result of characterization is also influenced by the IA32_HWP_REQUEST.Energy_Performance_Preference field, which must also be characterized.

Characterization can be used to set IA32_HWP_REQUEST.Minimum_Performance to achieve the required QOS in terms of performance. If IA32_HWP_REQUEST.Minimum_Performance is set higher than IA32_HWP_CAPABILITIES.Guaranteed_Performance then notification of excursions to Minimum Performance may be continuous.

If autonomous selection does not deliver the required workload performance, the OS should assess the current delivered effective frequency and for the duration of the specific performance requirement set IA32_HWP_REQUEST.Desired_Performance \neq 0 and adjust IA32_HWP_REQUEST.Energy_Performance_Preference as necessary to achieve the required workload performance. The MSR_PPERF.PCNT value can be used to better comprehend the potential performance result from adjustments to IA32_HWP_REQUEST.Desired_Performance. The OS should set IA32_HWP_REQUEST.Desired_Performance = 0 to re-enable autonomous selection.

Tuning for Maximum Performance or Lowest Power Consumption

Maximum performance will be delivered by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0 (performance preference).

Lowest power will be achieved by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0FFH (energy efficiency preference).

Mixing Logical Processor and Package Level HWP Field Settings

Using the IA32_HWP_REQUEST.Package_Control bit and the five valid bits in that MSR, the OS can mix and match between selecting the Logical Processor scope fields and the Package level fields. For example, the OS can set all logical cores' IA32_HWP_REQUEST.Package_Control bit to '1', and for those logical processors if it prefers a different EPP value than the one set in the IA32_HWP_REQUEST_PKG MSR, the OS can set the desired EPP value and the EPP valid bit. This overrides the package EPP value for only a subset of the logical processors in the package.

Additional Guidelines

Set IA32_HWP_REQUEST.Energy_Performance_Preference as appropriate for the platform's current mode of operation. For example, a mobile platforms' setting may be towards performance preference when on AC power and more towards energy efficiency when on DC power.

The use of the Running Average Power Limit (RAPL) processor capability (see section 14.7.1) is highly recommended when HWP is enabled. Use of IA32_HWP_Request.Maximum_Performance for thermal control is subject to limitations and can adversely impact the performance of other processor components e.g. Graphics

If default values deliver undesirable performance latency in response to events, the OS should set IA32_HWP_REQUEST.Activity_Window to a low (non-zero) value and IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) for the event duration.

Similarly, for “real-time” threads, set IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) and IA32_HWP_REQUEST.Activity_Window to a low value, e.g. 01H, for the duration of their execution.

When executing low priority work that may otherwise cause the hardware to deliver high performance, set IA32_HWP_REQUEST.Activity_Window to a longer value and reduce the IA32_HWP_Request.Maximum_Performance value as appropriate to control energy efficiency. Adjustments to IA32_HWP_REQUEST.Energy_Performance_Preference may also be necessary.

14.5 HARDWARE DUTY CYCLING (HDC)

Intel processors may contain support for Hardware Duty Cycling (HDC), which enables the processor to autonomously force its components inside the physical package into idle state. For example, the processor may selectively force only the processor cores into an idle state.

HDC is disabled by default on processors that support it. System software can dynamically enable or disable HDC to force one or more components into an idle state or wake up those components previously forced into an idle state. Forced Idling (and waking up) of multiple components in a physical package can be done with one WRMSR to a packaged-scope MSR from any logical processor within the same package.

HDC does not delay events such as timer expiration, but it may affect the latency of short (less than 1 msec) software threads, e.g. if a thread is forced to idle state just before completion and entering a “natural idle”.

HDC forced idle operation can be thought of as operating at a lower effective frequency. The effective average frequency computed by software will include the impact of HDC forced idle.

The primary use of HDC is enable system software to manage low active workloads to increase the package level C6 residency. Additionally, HDC can lower the effective average frequency in case of power or thermal limitation.

When HDC forces a logical processor, a processor core or a physical package to enter an idle state, its C-State is set to C3 or deeper. The deep “C-states” referred to in this section are processor-specific C-states.

14.5.1 Hardware Duty Cycling Programming Interfaces

The programming interfaces provided by HDC include the following:

- The CPUID instruction allows software to discover the presence of HDC support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input, bit 13 of EAX indicates the processor’s support of the following aspects of HDC.
 - Availability of HDC baseline resource, CPUID.06H:EAX[bit 13]: If this bit is set, HDC provides the following architectural MSRs: IA32_PKG_HDC_CTL, IA32_PM_CTL1, and the IA32_THREAD_STALL MSRs.
- Additionally, HDC may provide several non-architectural MSR.

Table 14-2. Architectural and non-Architecture MSRs Related to HDC

Address	Architectural	Register Name	Description
DB0H	Y	IA32_PKG_HDC_CTL	Package Enable/Disable HDC.
DB1H	Y	IA32_PM_CTL1	Per-logical-processor select control to allow/block HDC forced idling.
DB2H	Y	IA32_THREAD_STALL	Accumulate stalled cycles on this logical processor due to HDC forced idling.

Table 14-2. Architectural and non-Architecture MSRs Related to HDC

653H	N	MSR_CORE_HDC_RESIDENCY	Core level stalled cycle counter due to HDC forced idling on one or more logical processor.
655H	N	MSR_PKG_HDC_SHALLOW_RESIDENCY	Accumulate the cycles the package was in C2 ¹ state and at least one logical processor was in forced idle
656H	N	MSR_PKG_HDC_DEEP_RESIDENCY	Accumulate the cycles the package was in the software specified Cx ¹ state and at least one logical processor was in forced idle. Cx is specified in MSR_PKG_HDC_CONFIG_CTL.
652H	N	MSR_PKG_HDC_CONFIG_CTL	HDC configuration controls

NOTES:

1. The package “C-states” referred to in this section are processor-specific C-states.

14.5.2 Package level Enabling HDC

The layout of the IA32_PKG_HDC_CTL MSR is shown in Figure 14-16. IA32_PKG_HDC_CTL is a writable MSR from any logical processor in a package. The bit fields are described below:

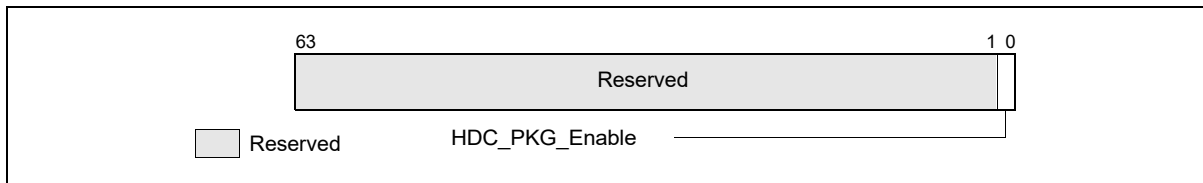


Figure 14-16. IA32_PKG_HDC_CTL MSR

- **HDC_PKG_Enable (bit 0, R/W)** — Software sets this bit to enable HDC operation by allowing the processor to force to idle all “HDC-allowed” (see Figure 14.5.3) logical processors in the package. Clearing this bit disables HDC operation in the package by waking up all the processor cores that were forced into idle by a previous ‘0’-to-‘1’ transition in IA32_PKG_HDC_CTL.HDC_PKG_Enable. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).
- Bits 63:1 are reserved and must be zero.

After processor support is determined via CPUID, system software can enable HDC operation by setting IA32_PKG_HDC_CTL.HDC_PKG_Enable to 1. At reset, IA32_PKG_HDC_CTL.HDC_PKG_Enable is cleared to 0. A ‘0’-to-‘1’ transition in HDC_PKG_Enable allows the processor to force to idle all HDC-allowed (indicated by the non-zero state of IA32_PM_CTL1[bit 0]) logical processors in the package. A ‘1’-to-‘0’ transition wakes up those HDC force-idled logical processors.

Software can enable or disable HDC using this package level control multiple times from any logical processor in the package. Note the latency of writing a value to the package-visible IA32_PKG_HDC_CTL.HDC_PKG_Enable is longer than the latency of a WRMSR operation to a Logical Processor MSR (as opposed to package level MSR) such as: IA32_PM_CTL1 (described in Section 14.5.3). Propagation of the change in IA32_PKG_HDC_CTL.HDC_PKG_Enable and reaching all HDC idled logical processor to be woken up may take on the order of core C6 exit latency.

14.5.3 Logical-Processor Level HDC Control

The layout of the IA32_PM_CTL1 MSR is shown in Figure 14-17. Each logical processor in a package has its own IA32_PM_CTL1 MSR. The bit fields are described below:

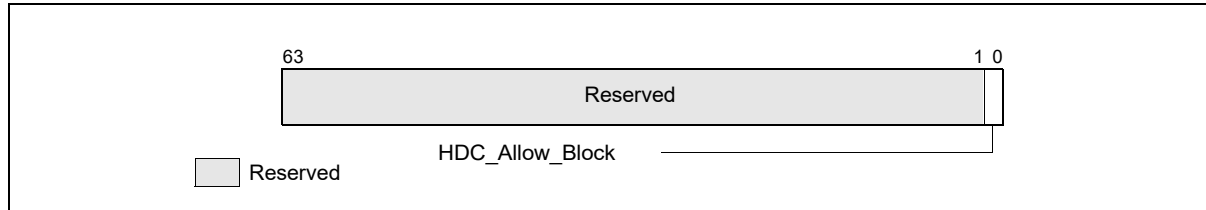


Figure 14-17. IA32_PM_CTL1 MSR

- **HDC_Allow_Block (bit 0, R/W)** — Software sets this bit to allow this logical processors to honor the package-level IA32_PKG_HDC_CTL.HDC_PKG_Enable control. Clearing this bit prevents this logical processor from using the HDC. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = one (1).
- Bits 63:1 are reserved and must be zero.

Fine-grain OS control of HDC operation at the granularity of per-logical-processor is provided by IA32_PM_CTL1. At RESET, all logical processors are allowed to participate in HDC operation such that OS can manage HDC using the package-level IA32_PKG_HDC_CTL.

Writes to IA32_PM_CTL1 complete with the latency that is typical to WRMSR to a Logical Processor level MSR. When the OS chooses to manage HDC operation at per-logical-processor granularity, it can write to IA32_PM_CTL1 on one or more logical processors as desired. Each write to IA32_PM_CTL1 must be done by code that executes on the logical processor targeted to be allowed into or blocked from HDC operation.

Blocking one logical processor for HDC operation may have package level impact. For example, the processor may decide to stop duty cycling of all other Logical Processors as well.

The propagation of IA32_PKG_HDC_CTL.HDC_PKG_Enable in a package takes longer than a WRMSR to IA32_PM_CTL1. The last completed write to IA32_PM_CTL1 on a logical processor will be honored when a '0'-to-'1' transition of IA32_PKG_HDC_CTL.HDC_PKG_Enable arrives to a logical processor.

14.5.4 HDC Residency Counters

There is a collection of counters available for software to track various residency metrics related to HDC operation. In general, HDC residency time is defined as the time in HDC forced idle state at the granularity of per-logical-processor, per-core, or package. At the granularity of per-core/package-level HDC residency, at least one of the logical processor in a core/package must be in the HDC forced idle state.

14.5.4.1 IA32_THREAD_STALL

Software can track per-logical-processor HDC residency using the architectural MSR IA32_THREAD_STALL. The layout of the IA32_THREAD_STALL MSR is shown in Figure 14-18. Each logical processor in a package has its own IA32_THREAD_STALL MSR. The bit fields are described below:

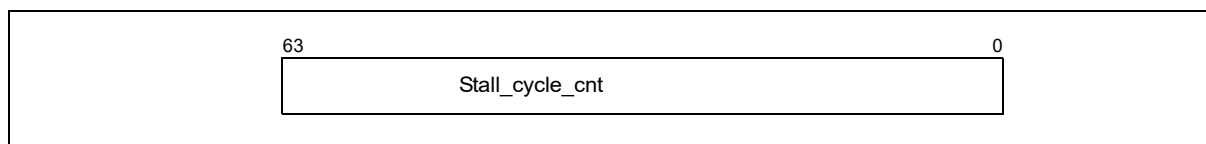


Figure 14-18. IA32_THREAD_STALL MSR

- **Stall_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after the logical processor exits from the forced idled C-state. At each update, the number of cycles that the logical processor was stalled due to forced-idle will be added to the counter. This counter is available only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).

A value of zero in IA32_THREAD_STALL indicates either HDC is not supported or the logical processor never serviced any forced HDC idle. A non-zero value in IA32_THREAD_STALL indicates the HDC forced-idle residency times of the logical processor. It also indicates the forced-idle cycles due to HDC that could appear as C0 time to traditional OS accounting mechanisms (e.g. time-stamping OS idle/exit events).

Software can read IA32_THREAD_STALL irrespective of the state of IA32_PKG_HDC_CTL and IA32_PM_CTL1, as long as CPUID.06H:EAX[bit 13] = 1.

14.5.4.2 Non-Architectural HDC Residency Counters

Processors that support HDC operation may provide the following model-specific HDC residency counters.

MSR_CORE_HDC_RESIDENCY

Software can track per-core HDC residency using the counter MSR_CORE_HDC_RESIDENCY. This counter increments when the core is in C3 state or deeper (all logical processors in this core are idle due to either HDC or other mechanisms) and at least one of the logical processors is in HDC forced idle state. The layout of the MSR_CORE_HDC_RESIDENCY is shown in Figure 14-19. Each processor core in a package has its own MSR_CORE_HDC_RESIDENCY MSR. The bit fields are described below:

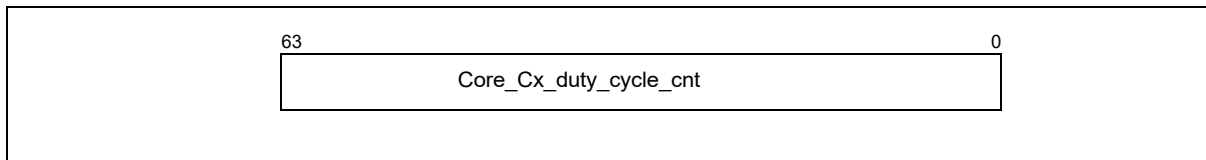


Figure 14-19. MSR_CORE_HDC_RESIDENCY MSR

- Core_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after core C-state exit from a forced idled C-state. At each update, the increment counts cycles when the core is in a Cx state (all its logical processor are idle) and at least one logical processor in this core was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR will cause a #GP fault. Default = zero (0).

A value of zero in MSR_CORE_HDC_RESIDENCY indicates either HDC is not supported or this processor core never serviced any forced HDC idle.

MSR_PKG_HDC_SHALLOW_RESIDENCY

The counter MSR_PKG_HDC_SHALLOW_RESIDENCY allows software to track HDC residency time when the package is in C2 state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. The layout of the MSR_PKG_HDC_SHALLOW_RESIDENCY is shown in Figure 14-20. There is one MSR_PKG_HDC_SHALLOW_RESIDENCY per package. The bit fields are described below:

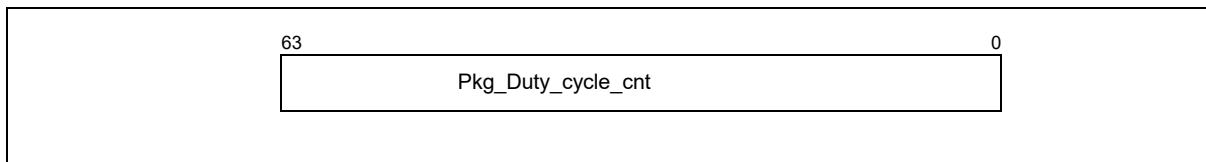


Figure 14-20. MSR_PKG_HDC_SHALLOW_RESIDENCY MSR

- Pkg_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. Package shallow residency may be implementation specific. In the initial implementation, the threshold is package C2-state. The count is updated only after package C2-state exit from a forced idled C-state. At each update, the increment counts cycles when

the package is in C2 state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

MSR_PKG_HDC_DEEP_RESIDENCY

The counter MSR_PKG_HDC_DEEP_RESIDENCY allows software to track HDC residency time when the package is in a software-specified package Cx state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. Selection of a specific package Cx state can be configured using MSR_PKG_HDC_CONFIG. The layout of the MSR_PKG_HDC_DEEP_RESIDENCY is shown in Figure 14-21. There is one MSR_PKG_HDC_DEEP_RESIDENCY per package. The bit fields are described below:

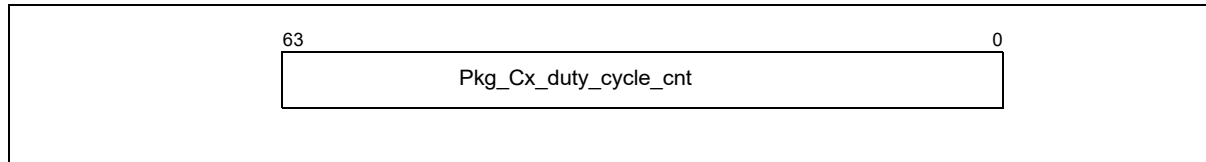


Figure 14-21. MSR_PKG_HDC_DEEP_RESIDENCY MSR

- **Pkg_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after package C-state exit from a forced idle state. At each update, the increment counts cycles when the package is in the software-configured Cx state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

MSR_PKG_HDC_CONFIG

MSR_PKG_HDC_CONFIG allows software to configure the package Cx state that the counter MSR_PKG_HDC_DEEP_RESIDENCY monitors. The layout of the MSR_PKG_HDC_CONFIG is shown in Figure 14-22. There is one MSR_PKG_HDC_CONFIG per package. The bit fields are described below:

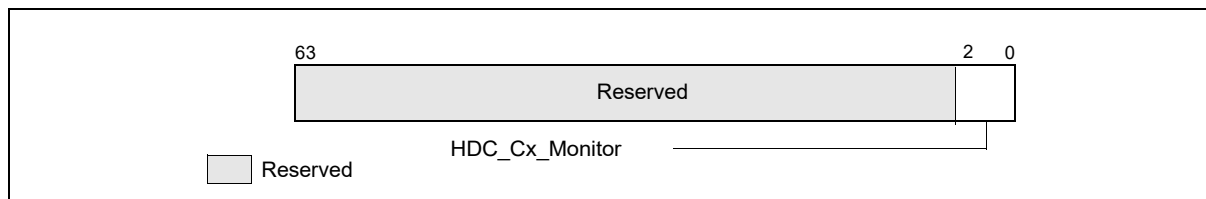


Figure 14-22. MSR_PKG_HDC_CONFIG MSR

- **Pkg_Cx_Monitor (bits 2:0, R/W)** — Selects which package C-state the MSR_HDC_DEEP_RESIDENCY counter will monitor. The encoding of the HDC_Cx_Monitor field are: **0**: no-counting; **1**: count package C2 only; **2**: count package C3 and deeper; **3**: count package C6 and deeper; **4**: count package C7 and deeper; other encodings are reserved. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).
- Bits 63:3 are reserved and must be zero.

14.5.5 MPERF and APERF Counters Under HDC

HDC operation can be thought of as an average effective frequency drop due to all or some of the Logical Processors enter an idle state period.

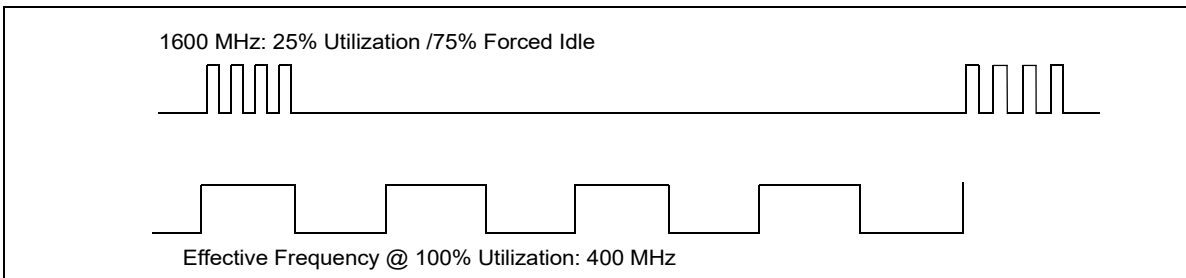


Figure 14-23. Example of Effective Frequency Reduction and Forced Idle Period of HDC

By default, the IA32_MPERF counter counts during forced idle periods as if the logical processor was active. The IA32_APERF counter does not count during forced idle state. This counting convention allows the OS to compute the average effective frequency of the Logical Processor between the last MWAIT exit and the next MWAIT entry (OS visible C0) by $\Delta\text{ACNT}/\Delta\text{MCNT} * \text{TSC Frequency}$.

14.6 HARDWARE FEEDBACK INTERFACE

Intel processors that enumerate CPUID.06H.0H:EAX.HW_FEEDBACK[bit 19] as 1 support Hardware Feedback Interface. Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a hardware feedback interface structure in memory. This structure has a global header that is 16 byte in size. Following this global header, there is one 8 byte entry per logical processor in the socket. The structure is designed as follows.

Table 14-3. Hardware Feedback Interface Structure

Byte Offset	Size (Bytes)	Description
0	16	Global Header
16	8	Per Logical Processor Entry
24	8	Per Logical Processor Entry
...
16 + n*8	8	Per Logical Processor Entry

The global header is structured as shown in Table 14-4.

Table 14-4. Hardware Feedback Interface Global Header Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	8	Timestamp	Timestamp of when the table was last updated by hardware. This is a timestamp in crystal clock units. Initialized by OS to 0.
8	1	Performance Capability Changed	If set to 1, indicates the performance capability field for one or more logical processors was updated in the table. Initialized by OS to 0.
9	1	Energy Efficiency Capability Changed	If set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table. Initialized by OS to 0.
10	6	Reserved	Initialized by OS to 0.

The per logical processor scheduler feedback entry is structured as follows. The operating system can determine the index of the logical processor feedback entry for a logical processor using CPUID.06H.0H:EDX[31:16] by executing CPUID on that logical processor.

Table 14-5. Hardware Feedback Interface Logical Processor Entry Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	1	Performance Capability	Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H.0H:EDX[0] enumerates support for Performance capability reporting.
1	1	Energy Efficiency Capability	Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H.0H:EDX[1] enumerates support for Energy Efficiency capability reporting.
2	6	Reserved	OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback.

14.6.1 Hardware Feedback Interface Pointer

The physical address of the hardware feedback interface structure is programmed by the OS into a package scoped MSR named IA32_HW_FEEDBACK_PTR. The MSR is structured as follows:

- Bits 63:MAXPHYADDR¹ - Reserved.
- Bits MAXPHYADDR-1:12 - ADDR. This is the physical address of the page frame of the first page of this structure.
- Bits 11:1 - Reserved.
- Bit 0 - Valid. When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.

The address of this MSR is 17D0H.

1. MAXPHYADDR is reported in CPUID.80000008H:EAX[7:0].

CPUID.06H.0H:EDX[11:8] enumerates the size of memory that must be allocated by the OS for this structure.

14.6.2 Hardware Feedback Interface Configuration

The operating system enables the hardware feedback interface using a package scoped MSR named IA32_HW_FEEDBACK_CONFIG (address 17D1H).

The MSR is structured as follows:

- Bits 63:1 - Reserved.
- Bit 0 - Enable. When set to 1, enables the hardware feedback interface.

Before enabling the hardware feedback interface, the OS must set a valid hardware feedback interface structure using IA32_HW_FEEDBACK_PTR.

When the Enable bit transitions from 1 to 0, hardware sets the IA32_PACKAGE_THERM_STATUS bit 26 to 1 to acknowledge disabling of the interface. The OS must wait for this bit to be set to 1 after disabling the interface before reclaiming the memory allocated for this structure. When this bit is set to 1, it is safe to reclaim the memory as it is guaranteed that there are no writes in progress to this structure by hardware.

Execution of GETSEC[SENDER] clears the enable bit to 0 on all sockets in the platform.

14.6.3 Hardware Feedback Interface Notifications

The IA32_PACKAGE_THERM_STATUS MSR is extended with a new bit, hardware feedback interface structure change status (bit 26, R/WC0), to indicate that the hardware has updated the hardware feedback interface structure. This is a sticky bit and once set, indicates that the OS should read the structure to determine the change and adjust its scheduling decisions. Once set, the hardware will not generate any further updates to this structure until the OS clears this bit by writing 0.

The OS can enable interrupt-based notifications when the structure is updated by hardware through a new enable bit, hardware feedback interrupt enable (bit 25, R/W), in the IA32_PACKAGE_THERM_INTERRUPT MSR. When this bit is set to 1, it enables the generation of an interrupt when the hardware feedback interface structure is updated by hardware. When the Enable bit transitions from 0 to 1, hardware will generate an initial notify, with the IA32_PACKAGE_THERM_STATUS bit 26 set to 1, to indicate that the OS should read the current hardware feedback interface structure.

14.7 MWAIT EXTENSIONS FOR ADVANCED POWER MANAGEMENT

IA-32 processors may support a number of C-states² that reduce power consumption for inactive states. Intel Core Solo and Intel Core Duo processors support both deeper C-state and MWAIT extensions that can be used by OS to implement power management policy.

Software should use CPUID to discover if a target processor supports the enumeration of MWAIT extensions. If CPUID.05H.ECX[Bit 0] = 1, the target processor supports MWAIT extensions and their enumeration (see Chapter 4, "Instruction Set Reference, M-U," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

If CPUID.05H.ECX[Bit 1] = 1, the target processor supports using interrupts as break-events for MWAIT, even when interrupts are disabled. Use this feature to measure C-state residency as follows:

- Software can write to bit 0 in the MWAIT Extensions register (ECX) when issuing an MWAIT to enter into a processor-specific C-state or sub C-state.
- When a processor comes out of an inactive C-state or sub C-state, software can read a timestamp before an interrupt service routine (ISR) is potentially executed.

2. The processor-specific C-states defined in MWAIT extensions can map to ACPI defined C-state types (C0, C1, C2, C3). The mapping relationship depends on the definition of a C-state by processor implementation and is exposed to OSPM by the BIOS using the ACPI defined _CST table.

CPUID.05H.EDX allows software to enumerate processor-specific C-states and sub C-states available for use with MWAIT extensions. IA-32 processors may support more than one C-state of a given C-state type. These are called sub C-states. Numerically higher C-state have higher power savings and latency (upon entering and exiting) than lower-numbered C-state.

At CPL = 0, system software can specify desired C-state and sub C-state by using the MWAIT hints register (EAX). Processors will not go to C-state and sub C-state deeper than what is specified by the hint register. If CPL > 0 and if MONITOR/MWAIT is supported at CPL > 0, the processor will only enter C1-state (regardless of the C-state request in the hints register).

Executing MWAIT generates an exception on processors operating at a privilege level where MONITOR/MWAIT are not supported.

NOTE

If MWAIT is used to enter a C-state (including sub C-state) that is numerically higher than C1, a store to the address range armed by MONITOR instruction will cause the processor to exit MWAIT if the store was originated by other processor agents. A store from non-processor agent may not cause the processor to exit MWAIT.

14.8 THERMAL MONITORING AND PROTECTION

The IA-32 architecture provides the following mechanisms for monitoring temperature and controlling thermal power:

1. The **catastrophic shutdown detector** forces processor execution to stop if the processor's core temperature rises above a preset limit.
2. **Automatic and adaptive thermal monitoring mechanisms** force the processor to reduce its power consumption in order to operate within predetermined temperature limits.
3. The **software controlled clock modulation mechanism** permits operating systems to implement power management policies that reduce power consumption; this is in addition to the reduction offered by automatic thermal monitoring mechanisms.
4. **On-die digital thermal sensor and interrupt mechanisms** permit the OS to manage thermal conditions natively without relying on BIOS or other system board components.

The first mechanism is not visible to software. The other three mechanisms are visible to software using processor feature information returned by executing CPUID with EAX = 1.

The second mechanism includes:

- **Automatic thermal monitoring** provides two modes of operation. One mode modulates the clock duty cycle; the second mode changes the processor's frequency. Both modes are used to control the core temperature of the processor.
- **Adaptive thermal monitoring** can provide flexible thermal management on processors made of multiple cores.

The third mechanism modulates the clock duty cycle of the processor. As shown in Figure 14-24, the phrase 'duty cycle' does not refer to the actual duty cycle of the clock signal. Instead it refers to the time period during which the clock signal is allowed to drive the processor chip. By using the stop clock mechanism to control how often the processor is clocked, processor power consumption can be modulated.

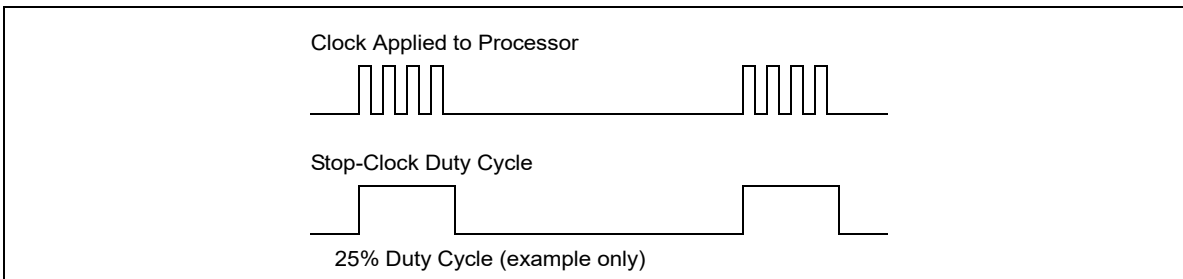


Figure 14-24. Processor Modulation Through Stop-Clock Mechanism

For previous automatic thermal monitoring mechanisms, software controlled mechanisms that changed processor operating parameters to impact changes in thermal conditions. Software did not have native access to the native thermal condition of the processor; nor could software alter the trigger condition that initiated software program control.

The fourth mechanism (listed above) provides access to an on-die digital thermal sensor using a model-specific register and uses an interrupt mechanism to alert software to initiate digital thermal monitoring.

14.8.1 Catastrophic Shutdown Detector

P6 family processors introduced a thermal sensor that acts as a catastrophic shutdown detector. This catastrophic shutdown detector was also implemented in Pentium 4, Intel Xeon and Pentium M processors. It is always enabled. When processor core temperature reaches a factory preset level, the sensor trips and processor execution is halted until after the next reset cycle.

14.8.2 Thermal Monitor

Pentium 4, Intel Xeon and Pentium M processors introduced a second temperature sensor that is factory-calibrated to trip when the processor's core temperature crosses a level corresponding to the recommended thermal design envelop. The trip-temperature of the second sensor is calibrated below the temperature assigned to the catastrophic shutdown detector.

14.8.2.1 Thermal Monitor 1

The Pentium 4 processor uses the second temperature sensor in conjunction with a mechanism called Thermal Monitor 1 (TM1) to control the core temperature of the processor. TM1 controls the processor's temperature by modulating the duty cycle of the processor clock. Modulation of duty cycles is processor model specific. Note that the processors STPCLK# pin is not used here; the stop-clock circuitry is controlled internally.

Support for TM1 is indicated by `CPUID.1:EDX.TM[bit 29] = 1`.

TM1 is enabled by setting the thermal-monitor enable flag (bit 3) in `IA32_MISC_ENABLE` [see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*]. Following a power-up or reset, the flag is cleared, disabling TM1. BIOS is required to enable only one automatic thermal monitoring modes. Operating systems and applications must not disable the operation of these mechanisms.

14.8.2.2 Thermal Monitor 2

An additional automatic thermal protection mechanism, called Thermal Monitor 2 (TM2), was introduced in the Intel Pentium M processor and also incorporated in newer models of the Pentium 4 processor family. Intel Core Duo and Solo processors, and Intel Core 2 Duo processor family all support TM1 and TM2. TM2 controls the core temperature of the processor by reducing the operating frequency and voltage of the processor and offers a higher performance level for a given level of power reduction than TM1.

TM2 is triggered by the same temperature sensor as TM1. The mechanism to enable TM2 may be implemented differently across various IA-32 processor families with different CPUID signatures in the family encoding value, but will be uniform within an IA-32 processor family.

Support for TM2 is indicated by CPUID.1:ECX.TM2[bit 8] = 1.

14.8.2.3 Two Methods for Enabling TM2

On processors with CPUID family/model/stepping signature encoded as 0x69n or 0x6Dn (early Pentium M processors), TM2 is enabled if the TM_SELECT flag (bit 16) of the MSR_THERM2_CTL register is set to 1 (Figure 14-25) and bit 3 of the IA32_MISC_ENABLE register is set to 1.

Following a power-up or reset, the TM_SELECT flag may be cleared. BIOS is required to enable either TM1 or TM2. Operating systems and applications must not disable mechanisms that enable TM1 or TM2. If bit 3 of the IA32_MISC_ENABLE register is set and TM_SELECT flag of the MSR_THERM2_CTL register is cleared, TM1 is enabled.

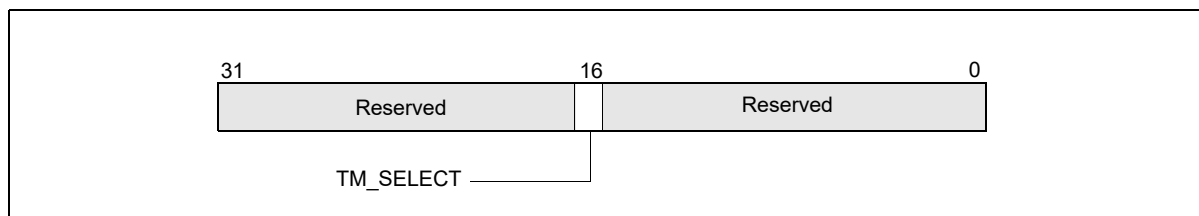


Figure 14-25. MSR_THERM2_CTL Register On Processors with CPUID Family/Model/Stepping Signature Encoded as 0x69n or 0x6Dn

On processors introduced after the Pentium 4 processor (this includes most Pentium M processors), the method used to enable TM2 is different. TM2 is enabled by setting bit 13 of IA32_MISC_ENABLE register to 1. This applies to Intel Core Duo, Core Solo, and Intel Core 2 processor family.

The target operating frequency and voltage for the TM2 transition after TM2 is triggered is specified by the value written to MSR_THERM2_CTL, bits 15:0 (Figure 14-26). Following a power-up or reset, BIOS is required to enable at least one of these two thermal monitoring mechanisms. If both TM1 and TM2 are supported, BIOS may choose to enable TM2 instead of TM1. Operating systems and applications must not disable the mechanisms that enable TM1 or TM2; and they must not alter the value in bits 15:0 of the MSR_THERM2_CTL register.

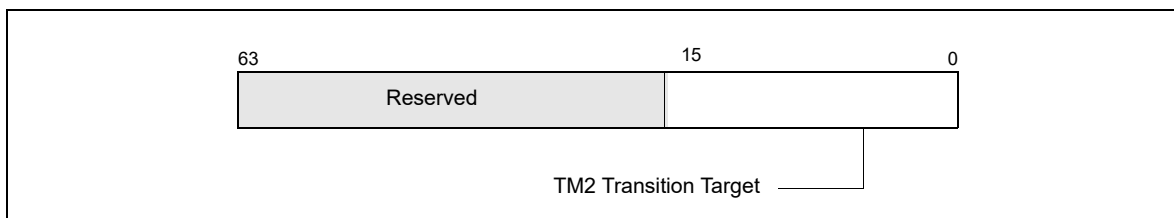


Figure 14-26. MSR_THERM2_CTL Register for Supporting TM2

14.8.2.4 Performance State Transitions and Thermal Monitoring

If the thermal control circuitry (TCC) for thermal monitor (TM1/TM2) is active, writes to the IA32_PERF_CTL will effect a new target operating point as follows:

- If TM1 is enabled and the TCC is engaged, the performance state transition can commence before the TCC is disengaged.

- If TM2 is enabled and the TCC is engaged, the performance state transition specified by a write to the IA32_PERF_CTL will commence after the TCC has disengaged.

14.8.2.5 Thermal Status Information

The status of the temperature sensor that triggers the thermal monitor (TM1/TM2) is indicated through the thermal status flag and thermal status log flag in the IA32_THERM_STATUS MSR (see Figure 14-27).

The functions of these flags are:

- **Thermal Status flag, bit 0** — When set, indicates that the processor core temperature is currently at the trip temperature of the thermal monitor and that the processor power consumption is being reduced via either TM1 or TM2, depending on which is enabled. When clear, the flag indicates that the core temperature is below the thermal monitor trip temperature. This flag is read only.
- **Thermal Status Log flag, bit 1** — When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

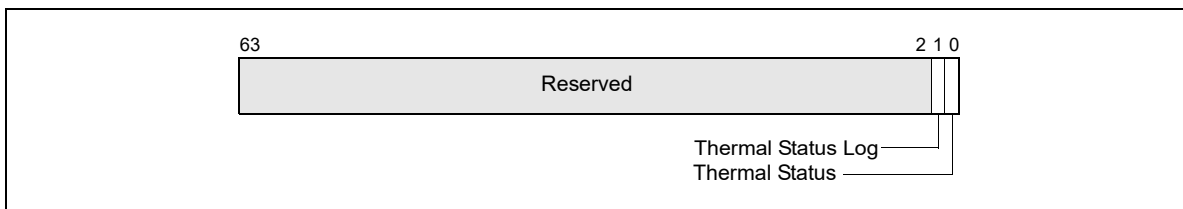


Figure 14-27. IA32_THERM_STATUS MSR

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for a minimum time period (on the order of 1 ms). The thermal monitor will remain engaged until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

While the processor is in a stop-clock state, interrupts will be blocked from interrupting the processor. This holding off of interrupts increases the interrupt latency, but does not cause interrupts to be lost. Outstanding interrupts remain pending until clock modulation is complete.

The thermal monitor can be programmed to generate an interrupt to the processor when the thermal sensor is tripped. The delivery mode, mask and vector for this interrupt can be programmed through the thermal entry in the local APIC's LVT (see Section 10.5.1, "Local Vector Table"). The low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR (see Figure 14-28) control when the interrupt is generated; that is, on a transition from a temperature below the trip point to above and/or vice-versa.

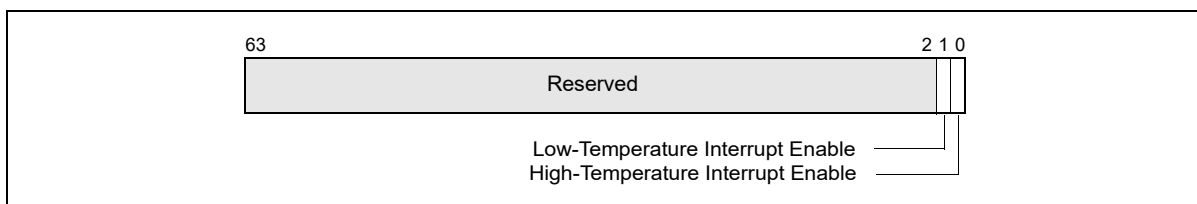


Figure 14-28. IA32_THERM_INTERRUPT MSR

- **High-Temperature Interrupt Enable flag, bit 0** — Enables an interrupt to be generated on the transition from a low-temperature to a high-temperature when set; disables the interrupt when clear.(R/W).
- **Low-Temperature Interrupt Enable flag, bit 1** — Enables an interrupt to be generated on the transition from a high-temperature to a low-temperature when set; disables the interrupt when clear.

The thermal monitor interrupt can be masked by the thermal LVT entry. After a power-up or reset, the low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR are

cleared (interrupts are disabled) and the thermal LVT entry is set to mask interrupts. This interrupt should be handled either by the operating system or system management mode (SMM) code.

Note that the operation of the thermal monitoring mechanism has no effect upon the clock rate of the processor's internal high-resolution timer (time stamp counter).

14.8.2.6 Adaptive Thermal Monitor

The Intel Core 2 Duo processor family supports enhanced thermal management mechanism, referred to as Adaptive Thermal Monitor (Adaptive TM).

Unlike TM2, Adaptive TM is not limited to one TM2 transition target. During a thermal trip event, Adaptive TM (if enabled) selects an optimal target operating point based on whether or not the current operating point has effectively cooled the processor.

Similar to TM2, Adaptive TM is enable by BIOS. The BIOS is required to test the TM1 and TM2 feature flags and enable all available thermal control mechanisms (including Adaptive TM) at platform initiation.

Adaptive TM is available only to a subset of processors that support TM2.

In each chip-multiprocessing (CMP) silicon die, each core has a unique thermal sensor that triggers independently. These thermal sensor can trigger TM1 or TM2 transitions in the same manner as described in Section 14.8.2.1 and Section 14.8.2.2. The trip point of the thermal sensor is not programmable by software since it is set during the fabrication of the processor.

Each thermal sensor in a processor core may be triggered independently to engage thermal management features. In Adaptive TM, both cores will transition to a lower frequency and/or lower voltage level if one sensor is triggered.

Triggering of this sensor is visible to software via the thermal interrupt LVT entry in the local APIC of a given core.

14.8.3 Software Controlled Clock Modulation

Pentium 4, Intel Xeon and Pentium M processors also support software-controlled clock modulation. This provides a means for operating systems to implement a power management policy to reduce the power consumption of the processor. Here, the stop-clock duty cycle is controlled by software through the IA32_CLOCK_MODULATION MSR (see Figure 14-29).

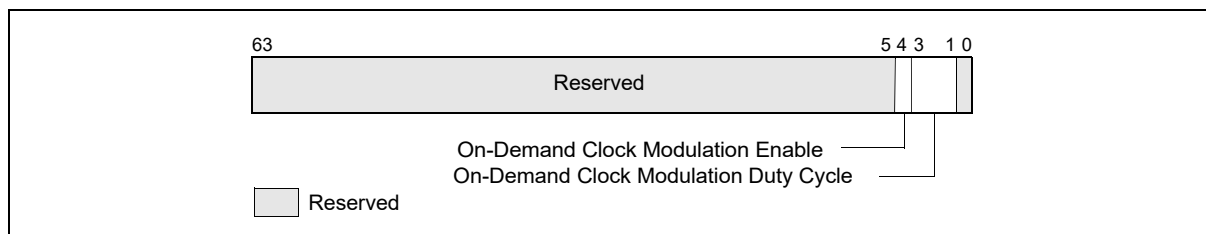


Figure 14-29. IA32_CLOCK_MODULATION MSR

The IA32_CLOCK_MODULATION MSR contains the following flag and field used to enable software-controlled clock modulation and to select the clock modulation duty cycle:

- **On-Demand Clock Modulation Enable, bit 4** — Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.
- **On-Demand Clock Modulation Duty Cycle, bits 1 through 3** — Selects the on-demand clock modulation duty cycle (see Table 14-6). This field is only active when the on-demand clock modulation enable flag is set.

Note that the on-demand clock modulation mechanism (like the thermal monitor) controls the processor's stop-clock circuitry internally to modulate the clock signal. The STPCLK# pin is not used in this mechanism.

Table 14-6. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

The on-demand clock modulation mechanism can be used to control processor power consumption. Power management software can write to the IA32_CLOCK_MODULATION MSR to enable clock modulation and to select a modulation duty cycle. If on-demand clock modulation and TM1 are both enabled and the thermal status of the processor is hot (bit 0 of the IA32_THERM_STATUS MSR is set), clock modulation at the duty cycle specified by TM1 takes precedence, regardless of the setting of the on-demand clock modulation duty cycle.

For Hyper-Threading Technology enabled processors, the IA32_CLOCK_MODULATION register is duplicated for each logical processor. In order for the On-demand clock modulation feature to work properly, the feature must be enabled on all the logical processors within a physical processor. If the programmed duty cycle is not identical for all the logical processors, the processor core clock will modulate to the highest duty cycle programmed for processors with any of the following CPUID DisplayFamily_DisplayModel signatures (see CPUID instruction in Chapter3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*): 06_1A, 06_1C, 06_1E, 06_1F, 06_25, 06_26, 06_27, 06_2C, 06_2E, 06_2F, 06_35, 06_36, and 0F_xx. For all other processors, if the programmed duty cycle is not identical for all logical processors in the same core, the processor core will modulate at the lowest programmed duty cycle.

For multiple processor cores in a physical package, each processor core can modulate to a programmed duty cycle independently.

For the P6 family processors, on-demand clock modulation was implemented through the chipset, which controlled clock modulation through the processor's STPCLK# pin.

14.8.3.1 Extension of Software Controlled Clock Modulation

Extension of the software controlled clock modulation facility supports on-demand clock modulation duty cycle with 4-bit dynamic range (increased from 3-bit range). Granularity of clock modulation duty cycle is increased to 6.25% (compared to 12.5%).

Four bit dynamic range control is provided by using bit 0 in conjunction with bits 3:1 of the IA32_CLOCK_MODULATION MSR (see Figure 14-30).

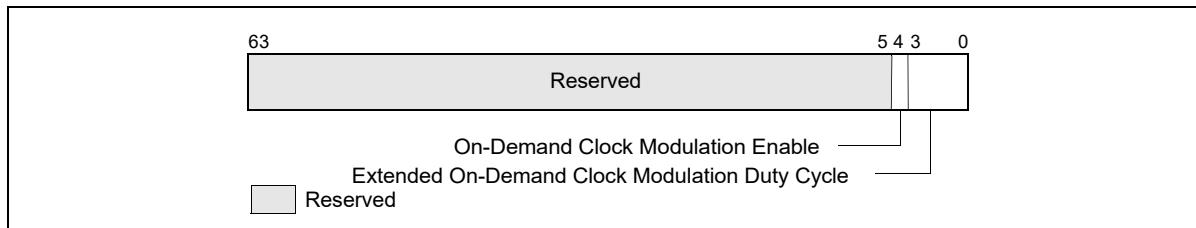


Figure 14-30. IA32_CLOCK_MODULATION MSR with Clock Modulation Extension

Extension to software controlled clock modulation is supported only if CPUID.06H:EAX[Bit 5] = 1. If CPUID.06H:EAX[Bit 5] = 0, then bit 0 of IA32_CLOCK_MODULATION is reserved.

14.8.4 Detection of Thermal Monitor and Software Controlled Clock Modulation Facilities

The ACPI flag (bit 22) of the CPUID feature flags indicates the presence of the IA32_THERM_STATUS, IA32_THERM_INTERRUPT, IA32_CLOCK_MODULATION MSRs, and the xAPIC thermal LVT entry.

The TM1 flag (bit 29) of the CPUID feature flags indicates the presence of the automatic thermal monitoring facilities that modulate clock duty cycles.

14.8.4.1 Detection of Software Controlled Clock Modulation Extension

Processor's support of software controlled clock modulation extension is indicated by CPUID.06H:EAX[Bit 5] = 1.

14.8.5 On Die Digital Thermal Sensors

On die digital thermal sensor can be read using an MSR (no I/O interface). In Intel Core Duo processors, each core has a unique digital sensor whose temperature is accessible using an MSR. The digital thermal sensor is the preferred method for reading the die temperature because (a) it is located closer to the hottest portions of the die, (b) it enables software to accurately track the die temperature and the potential activation of thermal throttling.

14.8.5.1 Digital Thermal Sensor Enumeration

The processor supports a digital thermal sensor if CPUID.06H:EAX[0] = 1. If the processor supports digital thermal sensor, EBX[bits 3:0] determine the number of thermal thresholds that are available for use.

Software sets thermal thresholds by using the IA32_THERM_INTERRUPT MSR. Software reads output of the digital thermal sensor using the IA32_THERM_STATUS MSR.

14.8.5.2 Reading the Digital Sensor

Unlike traditional analog thermal devices, the output of the digital thermal sensor is a temperature relative to the maximum supported operating temperature of the processor.

Temperature measurements returned by digital thermal sensors are always at or below TCC activation temperature. Critical temperature conditions are detected using the "Critical Temperature Status" bit. When this bit is set, the processor is operating at a critical temperature and immediate shutdown of the system should occur. Once the "Critical Temperature Status" bit is set, reliable operation is not guaranteed.

See Figure 14-31 for the layout of IA32_THERM_STATUS MSR. Bit fields include:

- **Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.
- **Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#). Bit 1 = 1 if PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **PROCHOT# or FORCEPR# Event (bit 2, RO)** — Indicates whether PROCHOT# or FORCEPR# is being asserted by another agent on the platform.

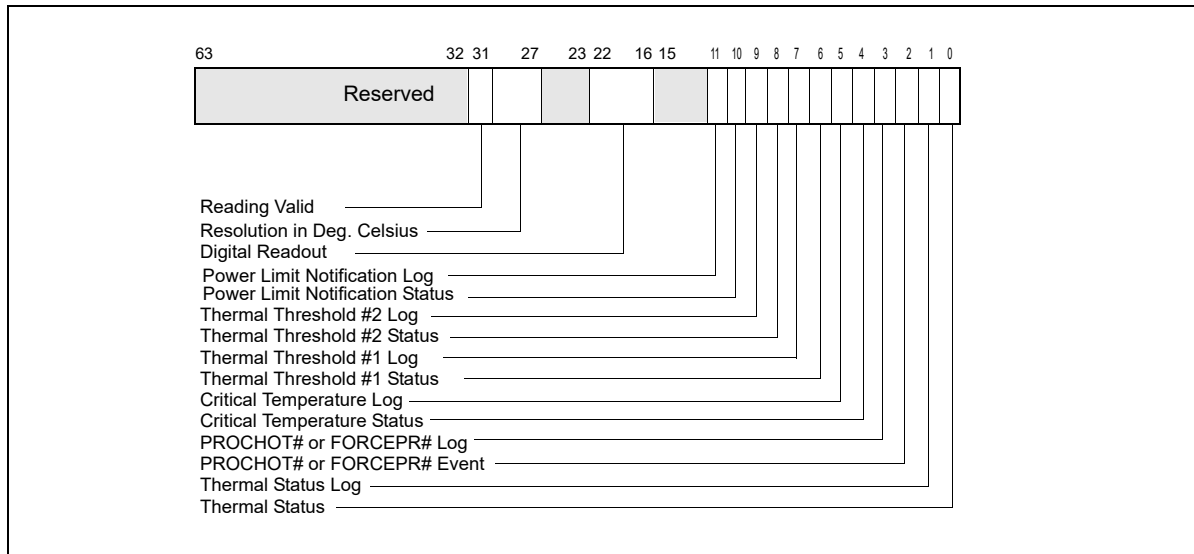


Figure 14-31. IA32_THERM_STATUS Register

- PROCHOT# or FORCEPR# Log (bit 3, R/WC0)** — Sticky bit that indicates whether PROCHOT# or FORCEPR# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, PROCHOT# or FORCEPR# has been externally asserted. Software may clear this bit by writing a zero. External PROCHOT# assertions are only acknowledged if the Bidirectional Prochot feature is enabled.
- Critical Temperature Status (bit 4, RO)** — Indicates whether the critical temperature detector output signal is currently active. If bit 4 = 1, the critical temperature detector output signal is currently active.
- Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.
- Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual temperature is currently higher than or equal to the value set in Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to TT#1. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.
- Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Threshold #1 has been reached. Software may clear this bit by writing a zero.
- Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual temperature is currently higher than or equal to the value set in Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to TT#2. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.
- Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.
- Power Limitation Status (bit 10, RO)** — Indicates whether the processor is currently operating below OS-requested P-state (specified in IA32_PERF_CTL) or OS-requested clock modulation duty cycle (specified in IA32_CLOCK_MODULATION). This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be delivered independently to IA32_PACKAGE_THERM_STATUS MSR.
- Power Notification Log (bit 11, R/WC0)** — Sticky bit that indicates the processor went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification is indicated independently in IA32_PACKAGE_THERM_STATUS MSR.

- **Digital Readout (bits 22:16, RO)** — Digital temperature reading in 1 degree Celsius relative to the TCC activation temperature.
0: TCC Activation temperature,
1: (TCC Activation - 1) , etc. See the processor's data sheet for details regarding TCC activation.
A lower reading in the Digital Readout field (bits 22:16) indicates a higher actual temperature.
- **Resolution in Degrees Celsius (bits 30:27, RO)** — Specifies the resolution (or tolerance) of the digital thermal sensor. The value is in degrees Celsius. It is recommended that new threshold values be offset from the current temperature by at least the resolution + 1 in order to avoid hysteresis of interrupt generation.
- **Reading Valid (bit 31, RO)** — Indicates if the digital readout in bits 22:16 is valid. The readout is valid if bit 31 = 1.

Changes to temperature can be detected using two thresholds (see Figure 14-32); one is set above and the other below the current temperature. These thresholds have the capability of generating interrupts using the core's local APIC which software must then service. Note that the local APIC entries used by these thresholds are also used by the Intel® Thermal Monitor; it is up to software to determine the source of a specific interrupt.

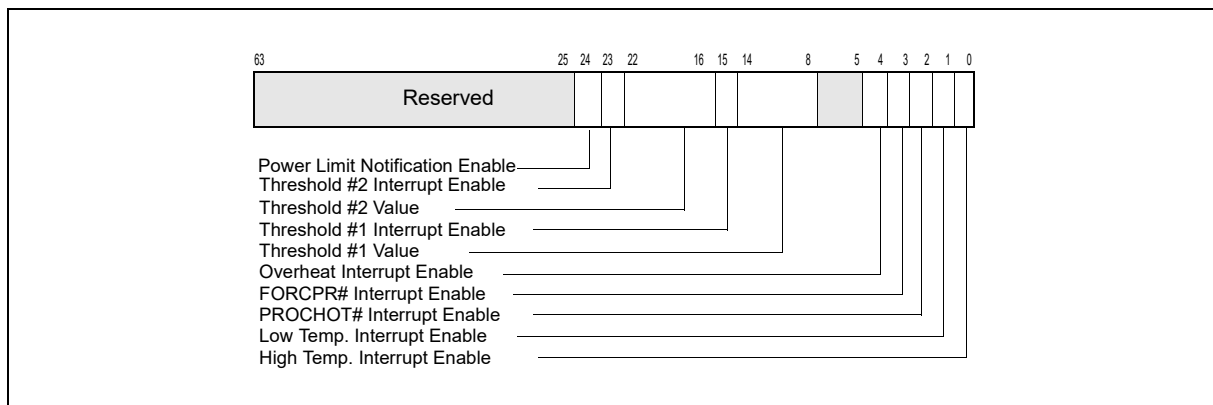


Figure 14-32. IA32_THERM_INTERRUPT Register

See Figure 14-32 for the layout of IA32_THERM_INTERRUPT MSR. Bit fields include:

- **High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.
- **Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.
- **PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.
- **FORCEPR# Interrupt Enable (bit 3, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when FORCEPR# has been asserted by another agent on the platform. Bit 3 = 0 disables the interrupt; bit 3 = 1 enables the interrupt.
- **Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.
- **Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #1 Status and Log bits as well as the Threshold #1 thermal interrupt delivery.

- **Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.
- **Threshold #2 Value (bits 22:16, R/W)** — A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #2 Status and Log bits as well as the Threshold #2 thermal interrupt delivery.
- **Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #2 setting in any direction. Bit 23 = 1 enables the interrupt; bit 23 = 0 disables the interrupt.
- **Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of power notification events when the processor went below OS-requested P-state or OS-requested clock modulation duty cycle. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be enabled independently by IA32_PACKAGE_THERM_INTERRUPT MSR.

14.8.6 Power Limit Notification

Platform firmware may be capable of specifying a power limit to restrict power delivered to a platform component, such as a physical processor package. This constraint imposed by platform firmware may occasionally cause the processor to operate below OS-requested P or T-state. A power limit notification event can be delivered using the existing thermal LVT entry in the local APIC.

Software can enumerate the presence of the processor's support for power limit notification by verifying CPUID.06H:EAX[bit 4] = 1.

If CPUID.06H:EAX[bit 4] = 1, then IA32_THERM_INTERRUPT and IA32_THERM_STATUS provides the following facility to manage power limit notification:

- Bits 10 and 11 in IA32_THERM_STATUS informs software of the occurrence of processor operating below OS-requested P-state or clock modulation duty cycle setting (see Figure 14-31).
- Bit 24 in IA32_THERM_INTERRUPT enables the local APIC to deliver a thermal event when the processor went below OS-requested P-state or clock modulation duty cycle setting (see Figure 14-32).

14.9 PACKAGE LEVEL THERMAL MANAGEMENT

The thermal management facilities like IA32_THERM_INTERRUPT and IA32_THERM_STATUS are often implemented with a processor core granularity. To facilitate software manage thermal events from a package level granularity, two architectural MSR is provided for package level thermal management. The IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT MSRs use similar interfaces as IA32_THERM_STATUS and IA32_THERM_INTERRUPT, but are shared in each physical processor package.

Software can enumerate the presence of the processor's support for package level thermal management facility (IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT) by verifying CPUID.06H:EAX[bit 6] = 1.

The layout of IA32_PACKAGE_THERM_STATUS MSR is shown in Figure 14-33.

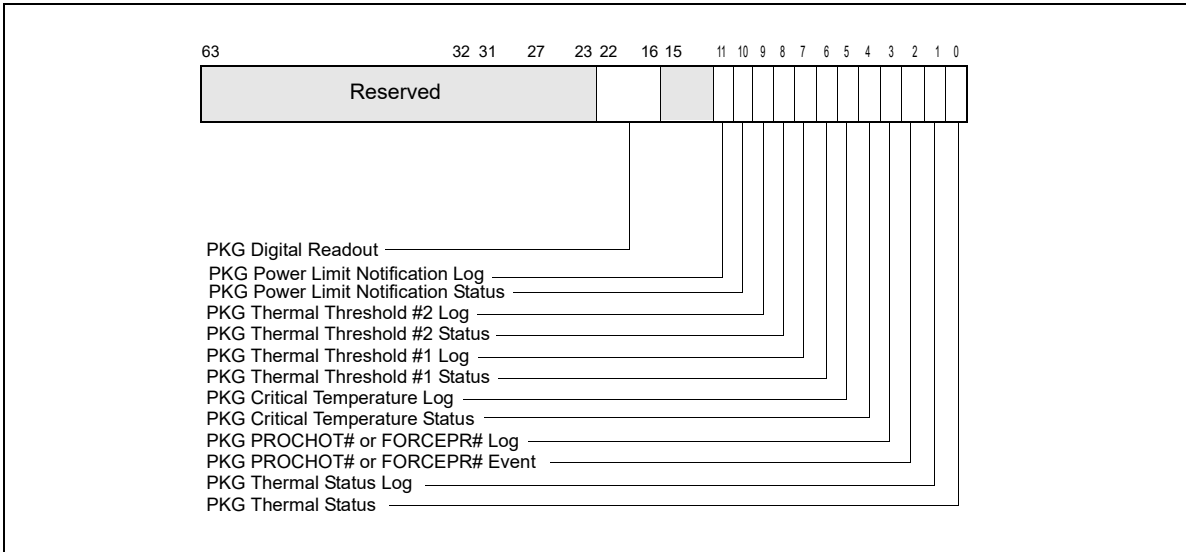


Figure 14-33. IA32_PACKAGE_THERM_STATUS Register

- **Package Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) for the package is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.
- **Package Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#) of the package. Bit 1 = 1 if package PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **Package PROCHOT# Event (bit 2, RO)** — Indicates whether package PROCHOT# is being asserted by another agent on the platform.
- **Package PROCHOT# Log (bit 3, R/WC0)** — Sticky bit that indicates whether package PROCHOT# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, package PROCHOT# has been externally asserted. Software may clear this bit by writing a zero.
- **Package Critical Temperature Status (bit 4, RO)** — Indicates whether the package critical temperature detector output signal is currently active. If bit 4 = 1, the package critical temperature detector output signal is currently active.
- **Package Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the package critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to PTT#1. Quantitative information of actual package temperature can be inferred from Package Digital Readout, bits 22:16.
- **Package Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Package Thermal Threshold #1 has been reached. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to PTT#2. Quantitative information of actual temperature can be inferred from Package Digital Readout, bits 22:16.
- **Package Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Package Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.

- **Package Power Limitation Status (bit 10, RO)** — Indicates package power limit is forcing one or more processors to operate below OS-requested P-state. Note that package power limit violation may be caused by processor cores or by devices residing in the uncore. Software can examine IA32_THERM_STATUS to determine if the cause originates from a processor core (see Figure 14-31).
- **Package Power Notification Log (bit 11, R/WCO)** — Sticky bit that indicates any processor in the package went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET.
- **Package Digital Readout (bits 22:16, RO)** — Package digital temperature reading in 1 degree Celsius relative to the package TCC activation temperature.
 - 0: Package TCC Activation temperature,
 - 1: (PTCC Activation - 1), etc. See the processor’s data sheet for details regarding PTCC activation.
 A lower reading in the Package Digital Readout field (bits 22:16) indicates a higher actual temperature.

The layout of IA32_PACKAGE_THERM_INTERRUPT MSR is shown in Figure 14-34.

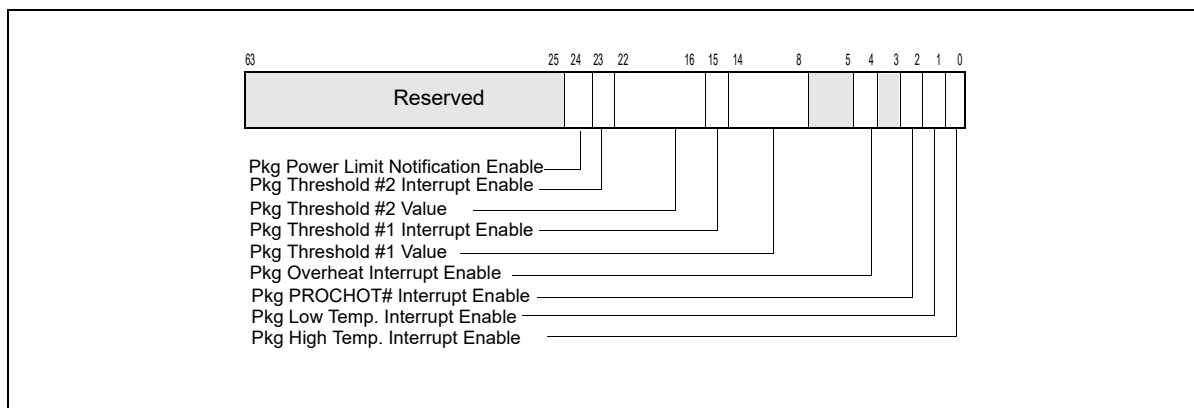


Figure 14-34. IA32_PACKAGE_THERM_INTERRUPT Register

- **Package High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a package high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.
- **Package Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.
- **Package PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when Package PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.
- **Package Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Package Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.
- **Package Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the Package TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #1 Status and Log bits as well as the Package Threshold #1 thermal interrupt delivery.
- **Package Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.
- **Package Threshold #2 Value (bits 22:16, R/W)** — A temperature threshold, encoded relative to the PTCC Activation temperature (using the same format as the Package Digital Readout). This threshold is compared

against the Package Digital Readout and is used to generate the Package Thermal Threshold #2 Status and Log bits as well as the Package Threshold #2 thermal interrupt delivery.

- **Package Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #2 setting in any direction. Bit 23 = 1 enables the interrupt; bit 23 = 0 disables the interrupt.
- **Package Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of package power notification events.

14.9.1 Support for Passive and Active cooling

Passive and active cooling may be controlled by the OS power management agent through ACPI control methods. On platforms providing package level thermal management facility described in the previous section, it is recommended that active cooling (FAN control) should be driven by measuring the package temperature using the IA32_PACKAGE_THERM_INTERRUPT MSR.

Passive cooling (frequency throttling) should be driven by measuring (a) the core and package temperatures, or (b) only the package temperature. If measured package temperature led the power management agent to choose which core to execute passive cooling, then all cores need to execute passive cooling. Core temperature is measured using the IA32_THERMAL_STATUS and IA32_THERMAL_INTERRUPT MSRs. The exact implementation details depend on the platform firmware and possible solutions include defining two different thermal zones (one for core temperature and passive cooling and the other for package temperature and active cooling).

14.10 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT

This section covers power management interfaces that are not architectural but addresses the power management needs of several platform specific components. Specifically, RAPL (Running Average Power Limit) interfaces provide mechanisms to enforce power consumption limit. Power limiting usages have specific usages in client and server platforms.

For client platform power limit control and for server platforms used in a data center, the following power and thermal related usages are desirable:

- Platform Thermal Management: Robust mechanisms to manage component, platform, and group-level thermals, either proactively or reactively (e.g., in response to a platform-level thermal trip point).
- Platform Power Limiting: More deterministic control over the system's power consumption, for example to meet battery life targets on rack-level or container-level power consumption goals within a datacenter.
- Power/Performance Budgeting: Efficient means to control the power consumed (and therefore the sustained performance delivered) within and across platforms.

The server and client usage models are addressed by RAPL interfaces, which expose multiple domains of power rationing within each processor socket. Generally, these RAPL domains may be viewed to include hierarchically:

- Package domain is the processor die.
- Memory domain includes the directly-attached DRAM; an additional power plane may constitute a separate domain.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each processor complex. Programming specific RAPL domain across multiple sockets is not supported.

14.10.1 RAPL Interfaces

RAPL interfaces consist of non-architectural MSRs. Each RAPL domain supports the following set of capabilities, some of which are optional as stated below.

- Power limit - MSR interfaces to specify power limit, time window; lock bit, clamp bit etc.
- Energy Status - Power metering interface providing energy consumption information.

- Perf Status (Optional) - Interface providing information on the performance effects (regression) due to power limits. It is defined as a duration metric that measures the power limit effect in the respective domain. The meaning of duration is domain specific.
- Power Info (Optional) - Interface providing information on the range of parameters for a given domain, minimum power, maximum power etc.
- Policy (Optional) - 4-bit priority information that is a hint to hardware for dividing budget between sub-domains in a parent domain.

Each of the above capabilities requires specific units in order to describe them. Power is expressed in Watts, Time is expressed in Seconds, and Energy is expressed in Joules. Scaling factors are supplied to each unit to make the information presented meaningful in a finite number of bits. Units for power, energy, and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR.

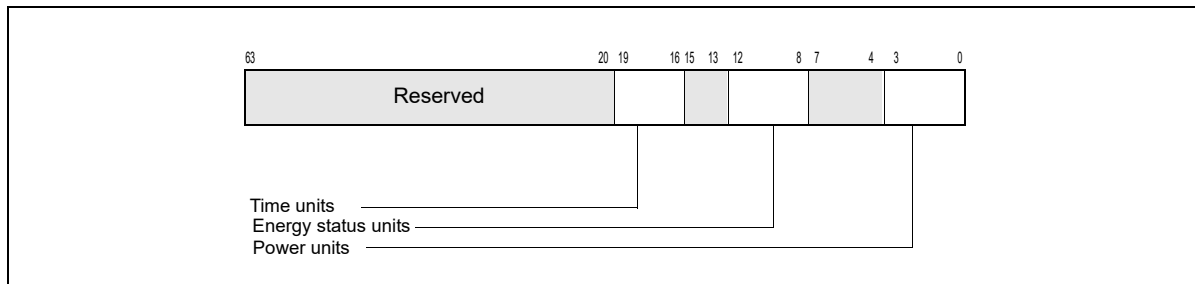


Figure 14-35. MSR_RAPL_POWER_UNIT Register

MSR_RAPL_POWER_UNIT (Figure 14-35) provides the following information across all RAPL domains:

- **Power Units** (bits 3:0): Power related information (in Watts) is based on the multiplier, $1/2^{PU}$; where PU is an unsigned integer represented by bits 3:0. Default value is 0011b, indicating power unit is in 1/8 Watts increment.
- **Energy Status Units** (bits 12:8): 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 10000b, indicating energy status unit is in 15.3 micro-Joules increment.
- **Time Units** (bits 19:16): Time related information (in Seconds) is based on the multiplier, $1/2^{TU}$; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating time unit is in 976 micro-seconds increment.

14.10.2 RAPL Domains and Platform Specificity

The specific RAPL domains available in a platform vary across product segments. Platforms targeting the client segment support the following RAPL domain hierarchy:

- Package
- Two power planes: PP0 and PP1 (PP1 may reflect to uncore devices)

Platforms targeting the server segment support the following RAPL domain hierarchy:

- Package
- Power plane: PPO
- DRAM

Each level of the RAPL hierarchy provides a respective set of RAPL interface MSRs. Table 14-7 lists the RAPL MSR interfaces available for each RAPL domain. The power limit MSR of each RAPL domain is located at offset 0 relative to an MSR base address which is non-architectural (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*). The energy status MSR of each domain is located at offset 1 relative to the MSR base address of respective domain.

Table 14-7. RAPL MSR Interfaces and RAPL Domains

Domain	Power Limit (Offset 0)	Energy Status (Offset 1)	Policy (Offset 2)	Perf Status (Offset 3)	Power Info (Offset 4)
PKG	MSR_PKG_POWER_LIMIT	MSR_PKG_ENERGY_STATUS	RESERVED	MSR_PKG_PERF_STATUS	MSR_PKG_POWER_INFO
DRAM	MSR_DRAM_POWER_LIMIT	MSR_DRAM_ENERGY_STATUS	RESERVED	MSR_DRAM_PERF_STATUS	MSR_DRAM_POWER_INFO
PP0	MSR_PP0_POWER_LIMIT	MSR_PP0_ENERGY_STATUS	MSR_PP0_POLICY	MSR_PP0_PERF_STATUS	RESERVED
PP1	MSR_PP1_POWER_LIMIT	MSR_PP1_ENERGY_STATUS	MSR_PP1_POLICY	RESERVED	RESERVED

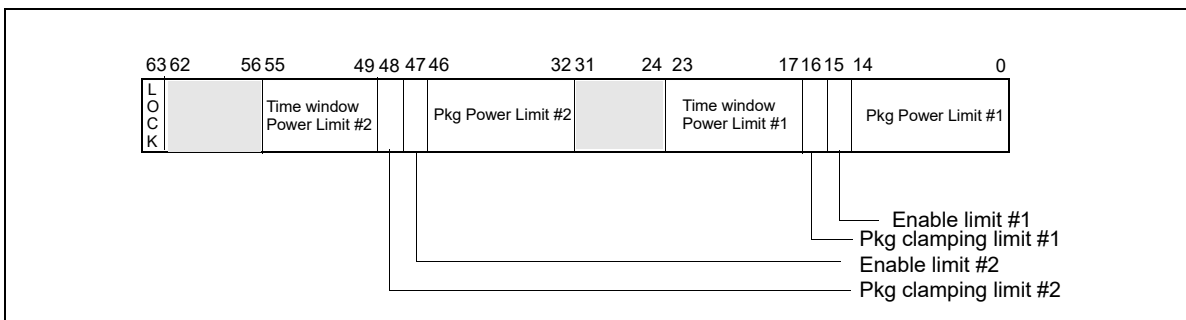
The presence of the optional MSR interfaces (the three right-most columns of Table 14-7) may be model-specific. See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for details.

14.10.3 Package RAPL Domain

The MSR interfaces defined for the package RAPL domain are:

- MSR_PKG_POWER_LIMIT allows software to set power limits for the package and measurement attributes associated with each limit,
- MSR_PKG_ENERGY_STATUS reports measured actual energy usage,
- MSR_PKG_POWER_INFO reports the package power range information for RAPL usage.

MSR_PKG_PERF_STATUS can report the performance impact of power limiting, but its availability may be model-specific.

**Figure 14-36. MSR_PKG_POWER_LIMIT Register**

MSR_PKG_POWER_LIMIT allows a software agent to define power limitation for the package domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_PKG_POWER_LIMIT. Two power limits can be specified, corresponding to time windows of different sizes. Each power limit provides independent clamping control that would permit the processor cores to go below OS-requested state to meet the power limits. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_PKG_POWER_LIMIT (Figure 14-36) are:

- **Package Power Limit #1** (bits 14:0): Sets the average power usage limit of the package domain corresponding to time window # 1. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.

- **Enable Power Limit #1**(bit 15): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #1** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #1** (bits 23:17): Indicates the time window for power limit #1
 $Time\ limit = 2^Y * (1.0 + Z/4.0) * Time_Unit$
 Here "Y" is the unsigned integer value represented. by bits 21:17, "Z" is an unsigned integer represented by bits 23:22. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Package Power Limit #2**(bits 46:32): Sets the average power usage limit of the package domain corresponding to time window # 2. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #2**(bit 47): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #2** (bit 48): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #2** (bits 55:49): Indicates the time window for power limit #2
 $Time\ limit = 2^Y * (1.0 + Z/4.0) * Time_Unit$
 Here "Y" is the unsigned integer value represented. by bits 53:49, "Z" is an unsigned integer represented by bits 55:54. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT. This field may have a hard-coded value in hardware and ignores values written by software.
- **Lock** (bit 63): If set, all write attempts to this MSR are ignored until next RESET.

MSR_PKG_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the package domain. This MSR is updated every ~1msec. It has a wraparound time of around 60 secs when power consumption is high, and may be longer otherwise.

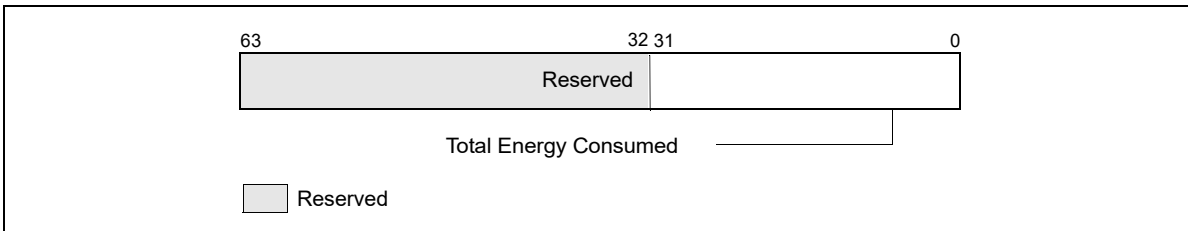


Figure 14-37. MSR_PKG_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_POWER_INFO is a read-only MSR. It reports the package power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the package domain. It also provides the largest possible time window for software to program the RAPL interface.

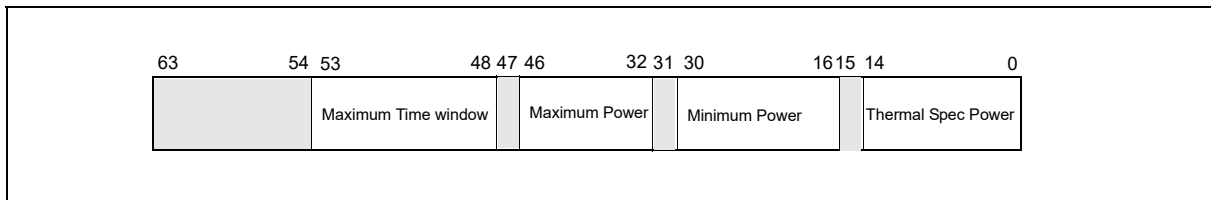


Figure 14-38. MSR_PKG_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): 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.

- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the package domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the package domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_PKG_POWER_LIMIT. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

MSR_PKG_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*).

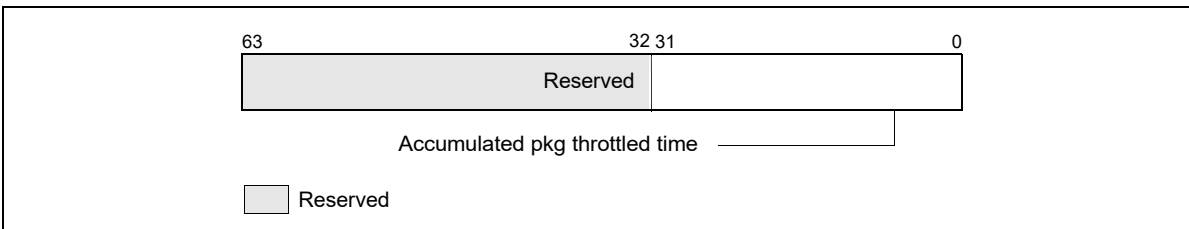


Figure 14-39. MSR_PKG_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the package has throttled. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

14.10.4 PP0/PP1 RAPL Domains

The MSR interfaces defined for the PP0 and PP1 domains are identical in layout. Generally, PP0 refers to the processor cores. The availability of PP1 RAPL domain interface is platform-specific. For a client platform, the PP1 domain refers to the power plane of a specific device in the uncore. For server platforms, the PP1 domain is not supported, but its PP0 domain supports the MSR_PP0_PERF_STATUS interface.

- MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow software to set power limits for the respective power plane domain.
- MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS report actual energy usage on a power plane.
- MSR_PP0_POLICY/MSR_PP1_POLICY allow software to adjust balance for respective power plane.

MSR_PP0_PERF_STATUS can report the performance impact of power limiting, but it is not available in client platforms.

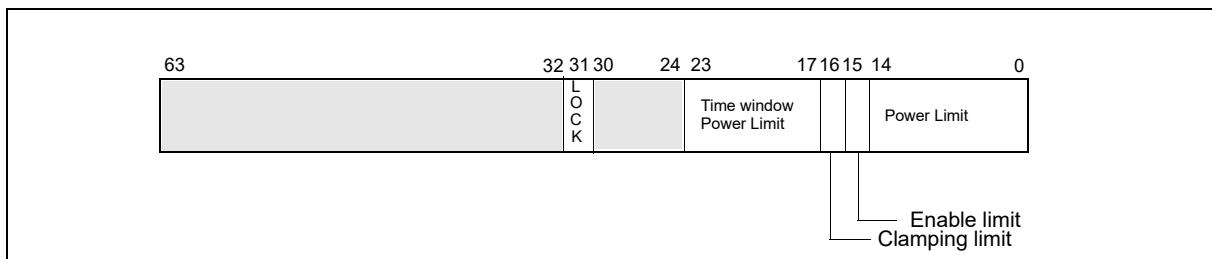


Figure 14-40. MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT Register

MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow a software agent to define power limitation for the respective power plane domain. A lock mechanism in each power plane domain allows the software agent to enforce power limit settings independently. Once a lock bit is set, the power limit settings in that power plane are static and un-modifiable until next RESET.

The bit fields of MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT (Figure 14-40) are:

- **Power Limit** (bits 14:0): Sets the average power usage limit of the respective power plane domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit** (bit 15): 0 = disabled; 1 = enabled.
- **Clamping Limitation** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit #1 will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of $2^Y * F$; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bit 31): If set, all write attempts to the MSR and corresponding policy MSR_PP0_POLICY/MSR_PP1_POLICY are ignored until next RESET.

MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS are read-only MSRs. They report the actual energy use for the respective power plane domains. These MSRs are updated every ~1msec.

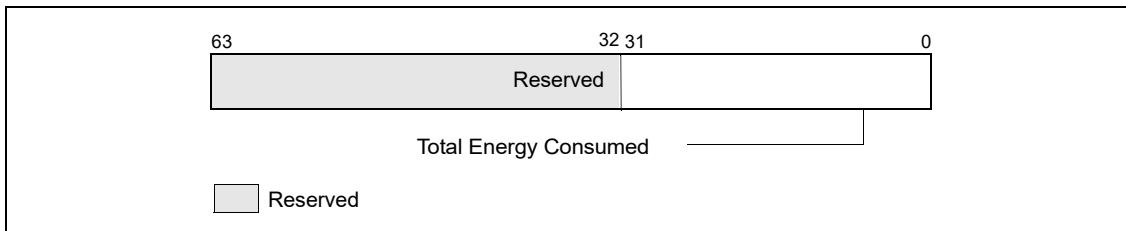


Figure 14-41. MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since the last time this register was cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PP0_POLICY/MSR_PP1_POLICY provide balance power policy control for each power plane by providing inputs to the power budgeting management algorithm. On platforms that support PP0 (IA cores) and PP1 (uncore graphic device), the default values give priority to the non-IA power plane. These MSRs enable the PCU to balance power consumption between the IA cores and uncore graphic device.

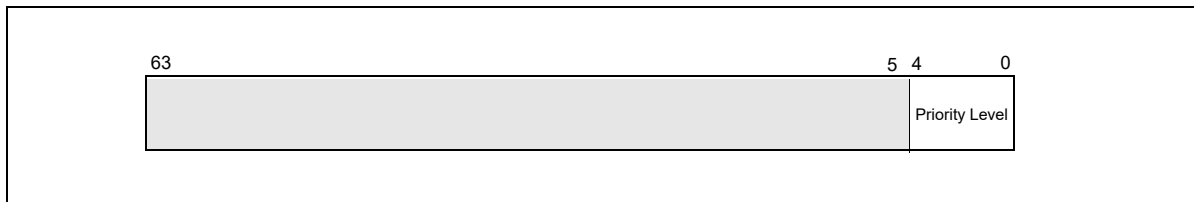


Figure 14-42. MSR_PP0_POLICY/MSR_PP1_POLICY Register

- **Priority Level** (bits 4:0): Priority level input to the PCU for respective power plane. PP0 covers the IA processor cores, PP1 covers the uncore graphic device. The value 31 is considered highest priority.

MSR_PP0_PERF_STATUS is a read-only MSR. It reports the total time for which the PP0 domain was throttled due to the power limits. This MSR is supported only in server platform. Throttling in this context is defined as going below the OS-requested P-state or T-state.

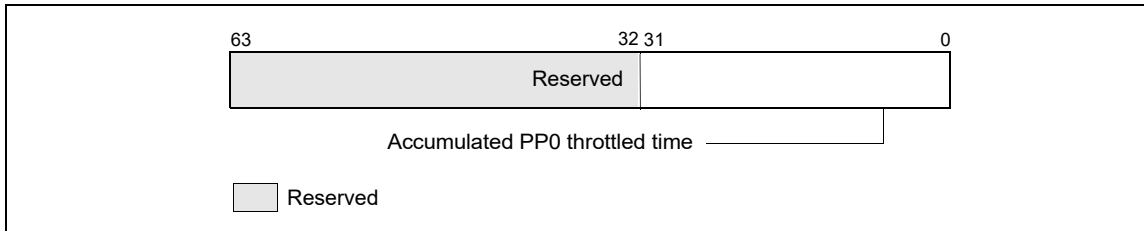


Figure 14-43. MSR_PPO_PERF_STATUS MSR

- **Accumulated PPO Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the PPO domain has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

14.10.5 DRAM RAPL Domain

The MSR interfaces defined for the DRAM domains are supported only in the server platform. The MSR interfaces are:

- MSR_DRAM_POWER_LIMIT allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.
- MSR_DRAM_ENERGY_STATUS reports measured actual energy usage.
- MSR_DRAM_POWER_INFO reports the DRAM domain power range information for RAPL usage.
- MSR_DRAM_PERF_STATUS can report the performance impact of power limiting.

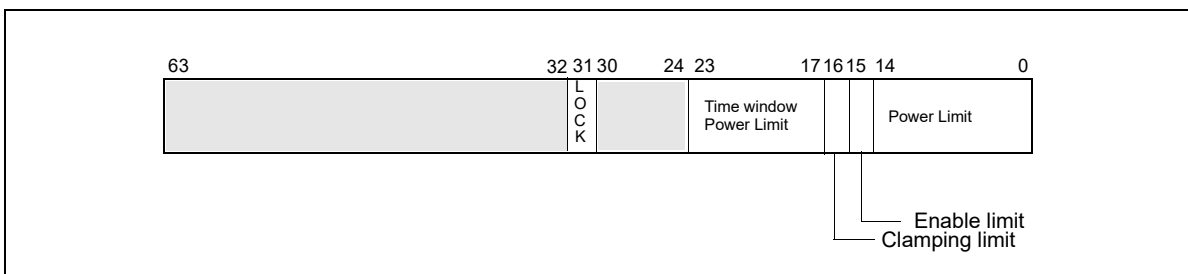


Figure 14-44. MSR_DRAM_POWER_LIMIT Register

MSR_DRAM_POWER_LIMIT allows a software agent to define power limitation for the DRAM domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_DRAM_POWER_LIMIT. A power limit can be specified along with a time window. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and unmodifiable until next RESET.

The bit fields of MSR_DRAM_POWER_LIMIT (Figure 14-44) are:

- **DRAM Power Limit #1** (bits 14:0): Sets the average power usage limit of the DRAM domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #1** (bit 15): 0 = disabled; 1 = enabled.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of $2^Y * F$; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bit 31): If set, all write attempts to this MSR are ignored until next RESET.

MSR_DRAM_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the DRAM domain. This MSR is updated every ~1msec.

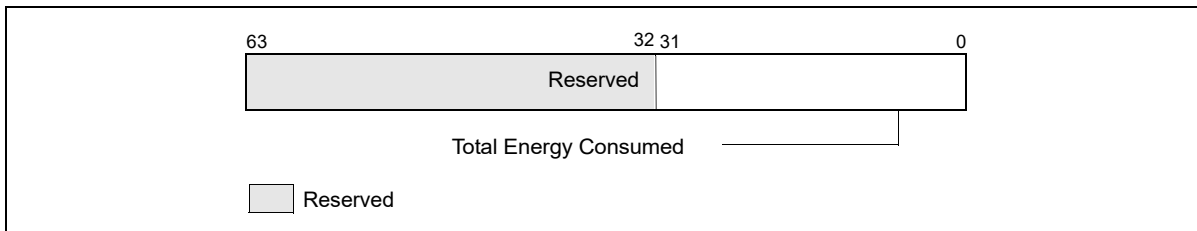


Figure 14-45. MSR_DRAM_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the “Energy Status Units” field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_POWER_INFO is a read-only MSR. It reports the DRAM power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the DRAM domain. It also provides the largest possible time window for software to program the RAPL interface.

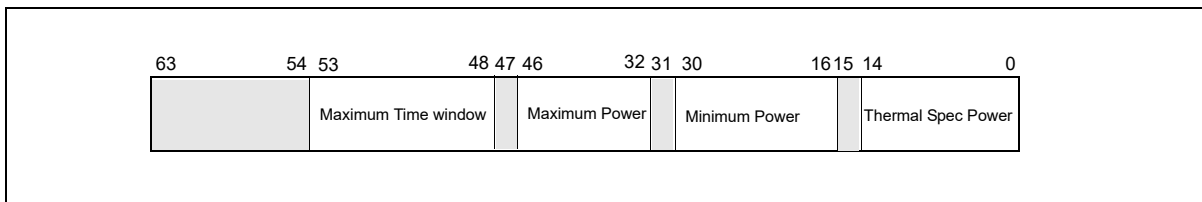


Figure 14-46. MSR_DRAM_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the DRAM domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the DRAM domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the DRAM domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_DRAM_POWER_LIMIT. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific (see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*).

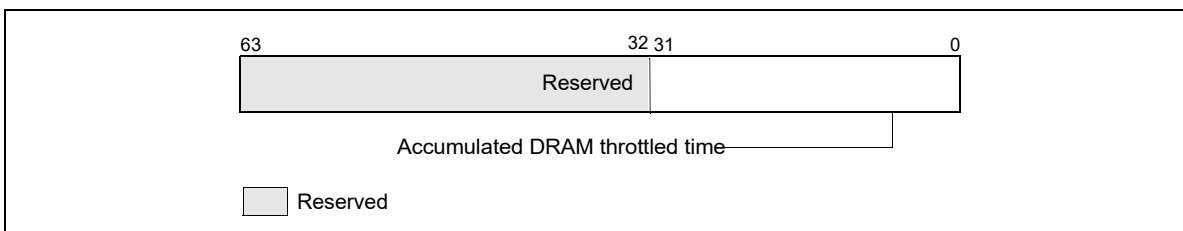


Figure 14-47. MSR_DRAM_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the DRAM domain has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

22. Updates to Chapter 15, Volume 3B

Change bars and green text show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

Changes to this chapter: Typo corrections.

This chapter describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC),” for more information on machine-check exceptions. A brief description of the Pentium processor’s machine check capability is also given.

Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

15.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH (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 processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 15.5 for detail.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 15.6 for detail.

15.2 COMPATIBILITY WITH PENTIUM PROCESSOR

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support and extend the machine-check exception mechanism introduced in the Pentium processor. The Pentium processor reports the following machine-check errors:

- data parity errors during read cycles
- unsuccessful completion of a bus cycle

The above errors are reported using the P5_MC_TYPE and P5_MC_ADDR MSRs (implementation specific for the Pentium processor). Use the RDMSR instruction to read these MSRs. See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for the addresses.

The machine-check error reporting mechanism that Pentium processors use is similar to that used in Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. When an error is detected, it is recorded in P5_MC_TYPE and P5_MC_ADDR; the processor then generates a machine-check exception (#MC).

See Section 15.3.3, “Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture,” and Section 15.10.2, “Pentium Processor Machine-Check Exception Handling,” for information on compatibility between machine-check code written to run on the Pentium processors and code written to run on P6 family processors.

15.3 MACHINE-CHECK MSRS

Machine check MSRs in the Pentium 4, Intel Atom, Intel Xeon, and P6 family processors consist of a set of global control and status registers and several error-reporting register banks. See Figure 15-1.

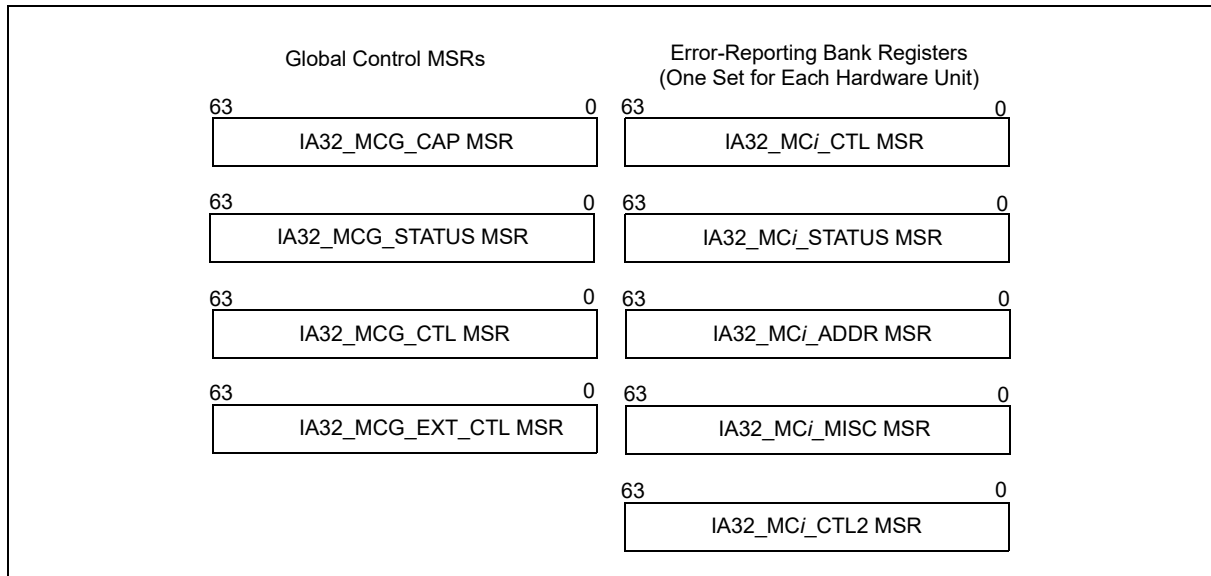


Figure 15-1. Machine-Check MSRs

Each error-reporting bank is associated with a specific hardware unit (or group of hardware units) in the processor. Use RDMSR and WRMSR to read and to write these registers.

15.3.1 Machine-Check Global Control MSRs

The machine-check global control MSRs include the IA32_MCG_CAP, IA32_MCG_STATUS, and optionally IA32_MCG_CTL and IA32_MCG_EXT_CTL. See Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for the addresses of these registers.

15.3.1.1 IA32_MCG_CAP MSR

The IA32_MCG_CAP MSR is a read-only register that provides information about the machine-check architecture of the processor. Figure 15-2 shows the layout of the register.

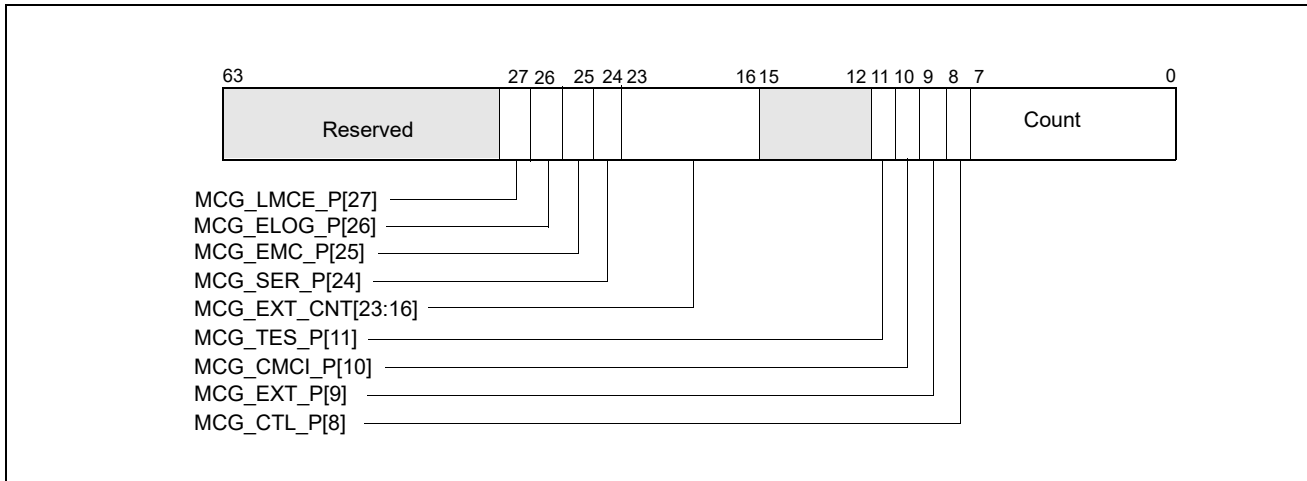


Figure 15-2. IA32_MCG_CAP Register

Where:

- **Count field, bits 7:0** — Indicates the number of hardware unit error-reporting banks available in a particular processor implementation.
- **MCG_CTL_P (control MSR present) flag, bit 8** — Indicates that the processor implements the IA32_MCG_CTL MSR when set; this register is absent when clear.
- **MCG_EXT_P (extended MSRs present) flag, bit 9** — Indicates that the processor implements the extended machine-check state registers found starting at MSR address 180H; these registers are absent when clear.
- **MCG_CMCI_P (Corrected MC error counting/signaling extension present) flag, bit 10** — Indicates (when set) that extended state and associated MSRs necessary to support the reporting of an interrupt on a corrected MC error event and/or count threshold of corrected MC errors, is present. When this bit is set, it does not imply this feature is supported across all banks. Software should check the availability of the necessary logic on a bank by bank basis when using this signaling capability (i.e. bit 30 settable in individual IA32_MCi_CTL2 register).
- **MCG_TES_P (threshold-based error status present) flag, bit 11** — Indicates (when set) that bits 56:53 of the IA32_MCi_STATUS MSR are part of the architectural space. Bits 56:55 are reserved, and bits 54:53 are used to report threshold-based error status. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific.
- **MCG_EXT_CNT, bits 23:16** — Indicates the number of extended machine-check state registers present. This field is meaningful only when the MCG_EXT_P flag is set.
- **MCG_SER_P (software error recovery support present) flag, bit 24** — Indicates (when set) that the processor supports software error recovery (see Section 15.6), and IA32_MCi_STATUS MSR bits 56:55 are used to report the signaling of uncorrected recoverable errors and whether software must take recovery actions for uncorrected errors. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific. If MCG_TES_P is set but MCG_SER_P is not set, bits 56:55 are reserved.
- **MCG_EMCP (Enhanced Machine Check Capability) flag, bit 25** — Indicates (when set) that the processor supports enhanced machine check capabilities for firmware first signaling.
- **MCG_ELOG_P (extended error logging) flag, bit 26** — Indicates (when set) 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.

For additional information about extended error logging interface, see

<https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enhanced-mca-logging-xeon-paper.pdf>.

- **MCG_LMCE_P (local machine check exception) flag, bit 27** — Indicates (when set) that the following interfaces are present:
 - an extended state LMCE_S (located in bit 3 of IA32_MCG_STATUS), and
 - the IA32_MCG_EXT_CTL MSR, necessary to support Local Machine Check Exception (LMCE).

A non-zero MCG_LMCE_P indicates that, when LMCE is enabled as described in Section 15.3.1.5, some machine check errors may be delivered to only a single logical processor.

The effect of writing to the IA32_MCG_CAP MSR is undefined.

15.3.1.2 IA32_MCG_STATUS MSR

The IA32_MCG_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 15-3).

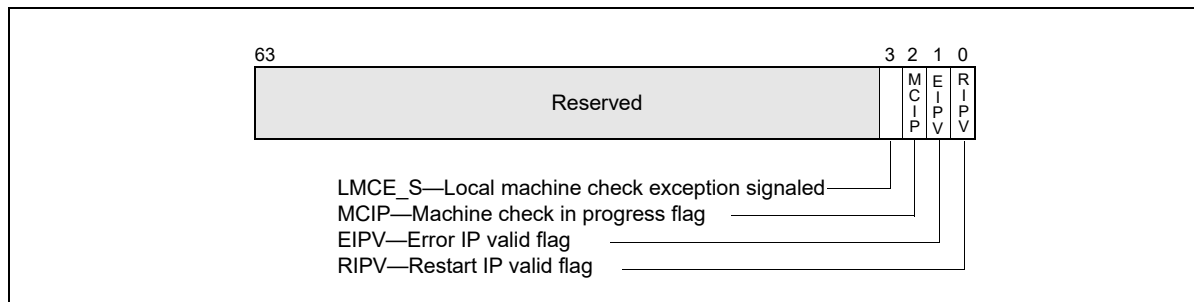


Figure 15-3. IA32_MCG_STATUS Register

Where:

- **RIPV (restart IP valid) flag, bit 0** — Indicates (when set) that program execution can be restarted reliably at the instruction pointed to by the instruction pointer pushed on the stack when the machine-check exception is generated. When clear, the program cannot be reliably restarted at the pushed instruction pointer.
- **EIPV (error IP valid) flag, bit 1** — Indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- **MCIP (machine check in progress) flag, bit 2** — Indicates (when set) that a machine-check exception was generated. Software can set or clear this flag. The occurrence of a second Machine-Check Event while MCIP is set will cause the processor to enter a shutdown state. For information on processor behavior in the shutdown state, please refer to the description in Chapter 6, "Interrupt and Exception Handling": "Interrupt 8—Double Fault Exception (#DF)".
- **LMCE_S (local machine check exception signaled), bit 3** — Indicates (when set) that a local machine-check exception was generated. This indicates that the current machine-check event was delivered to only this logical processor.

Bits 63:04 in IA32_MCG_STATUS are reserved. An attempt to write to IA32_MCG_STATUS with any value other than 0 would result in #GP.

15.3.1.3 IA32_MCG_CTL MSR

The IA32_MCG_CTL MSR is present if the capability flag MCG_CTL_P is set in the IA32_MCG_CAP MSR.

IA32_MCG_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features. All other values are undefined and/or implementation specific.

15.3.1.4 IA32_MCG_EXT_CTL MSR

The IA32_MCG_EXT_CTL MSR is present if the capability flag MCG_LMCE_P is set in the IA32_MCG_CAP MSR. IA32_MCG_EXT_CTL.LMCE_EN (bit 0) allows the processor to signal some MCEs to only a single logical processor in the system.

If MCG_LMCE_P is not set in IA32_MCG_CAP, or platform software has not enabled LMCE by setting IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20), any attempt to write or read IA32_MCG_EXT_CTL will result in #GP.

The IA32_MCG_EXT_CTL MSR is cleared on RESET.

Figure 15-4 shows the layout of the IA32_MCG_EXT_CTL register

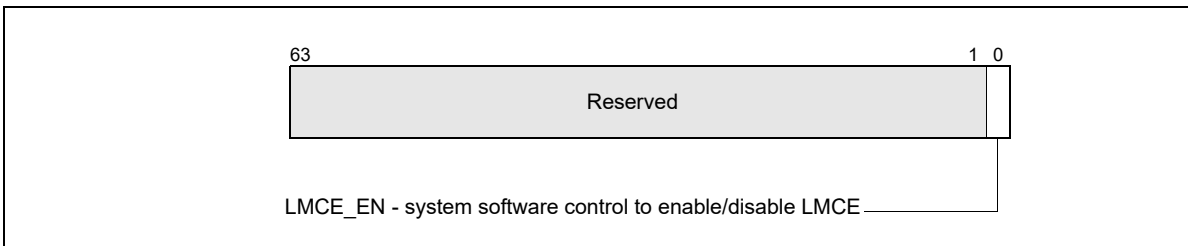


Figure 15-4. IA32_MCG_EXT_CTL Register

where

- **LMCE_EN (local machine check exception enable) flag, bit 0** - System software sets this to allow hardware to signal some MCEs to only a single logical processor. System software can set LMCE_EN only if the platform software has configured IA32_FEATURE_CONTROL as described in Section 15.3.1.5.

15.3.1.5 Enabling Local Machine Check

The intended usage of LMCE requires proper configuration by both platform software and system software. Platform software can turn LMCE on by setting bit 20 (LMCE_ENABLED) in IA32_FEATURE_CONTROL MSR (MSR address 3AH).

System software must ensure that both IA32_FEATURE_CONTROL.Lock (bit 0) and IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20) are set before attempting to set IA32_MCG_EXT_CTL.LMCE_EN (bit 0). When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor. Software should make no assumptions about the type of error that hardware can choose to deliver as LMCE. The severity and override rules stay the same as described in Table 15-8 to determine the recovery actions.

15.3.2 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32_MCi_CTL, IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32_MCG_CAP MSR (address 0179H). The first error-reporting register (IA32_MC0_CTL) always starts at address 400H.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for addresses of the error-reporting registers in the Pentium 4, Intel Atom, and Intel Xeon processors; and for addresses of the error-reporting registers P6 family processors.

15.3.2.1 IA32_MCi_CTL MSRs

The IA32_MCi_CTL MSR controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE_j) represents a potential error. Setting an EE_j flag enables signaling #MC of the associated error and clearing it disables signaling of the error. Error logging happens regardless of the setting

of these bits. The processor drops writes to bits that are not implemented. Figure 15-5 shows the bit fields of IA32_MCi_CTL.

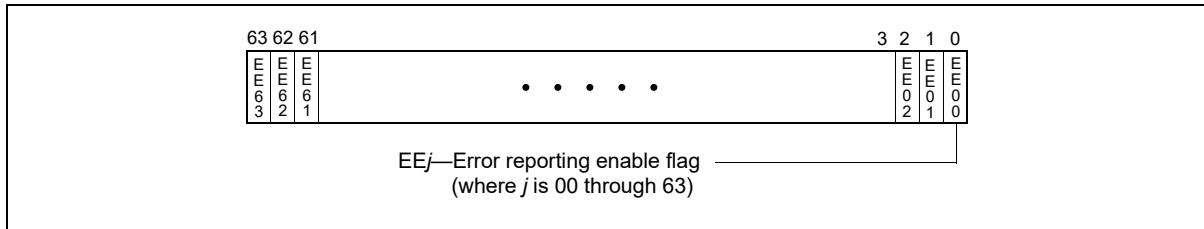


Figure 15-5. IA32_MCi_CTL Register

NOTE

For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH and onward): the operating system or executive software must not modify the contents of the IA32_MCO_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32_MCO_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32_MCi_CTL MSR.

15.3.2.2 IA32_MCi_STATUS MSRS

Each IA32_MCi_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 15-6). Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

NOTE

Figure 15-6 depicts the IA32_MCi_STATUS MSR when IA32_MCG_CAP[24] = 1, IA32_MCG_CAP[11] = 1 and IA32_MCG_CAP[10] = 1. When IA32_MCG_CAP[24] = 0 and IA32_MCG_CAP[11] = 1, bits 56:55 is reserved and bits 54:53 for threshold-based error reporting. When IA32_MCG_CAP[11] = 0, bits 56:53 are part of the “Other Information” field. The use of bits 54:53 for threshold-based error reporting began with Intel Core Duo processors, and is currently used for cache memory. See Section 15.4, “Enhanced Cache Error reporting,” for more information. When IA32_MCG_CAP[10] = 0, bits 52:38 are part of the “Other Information” field. The use of bits 52:38 for corrected MC error count is introduced with Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH.

Where:

- **MCA (machine-check architecture) error code field, bits 15:0** — Specifies the machine-check architecture-defined error code for the machine-check error condition detected. The machine-check architecture-defined error codes are guaranteed to be the same for all IA-32 processors that implement the machine-check architecture. See Section 15.9, “Interpreting the MCA Error Codes,” and Chapter 16, “Interpreting Machine-Check Error Codes”, for information on machine-check error codes.
- **Model-specific error code field, bits 31:16** — Specifies the model-specific error code that uniquely identifies the machine-check error condition detected. The model-specific error codes may differ among IA-32 processors for the same machine-check error condition. See Chapter 16, “Interpreting Machine-Check Error Codes” for information on model-specific error codes.
- **Reserved, Error Status, and Other Information fields, bits 56:32** —
 - If IA32_MCG_CAP.MCG EMC_P[bit 25] is 0, bits 37:32 contain “Other Information” that is implementation-specific and is not part of the machine-check architecture.
 - If IA32_MCG_CAP.MCG EMC_P is 1, “Other Information” is in bits 36:32. If bit 37 is 0, system firmware has not changed the contents of IA32_MCi_STATUS. If bit 37 is 1, system firmware may have edited the contents of IA32_MCi_STATUS.

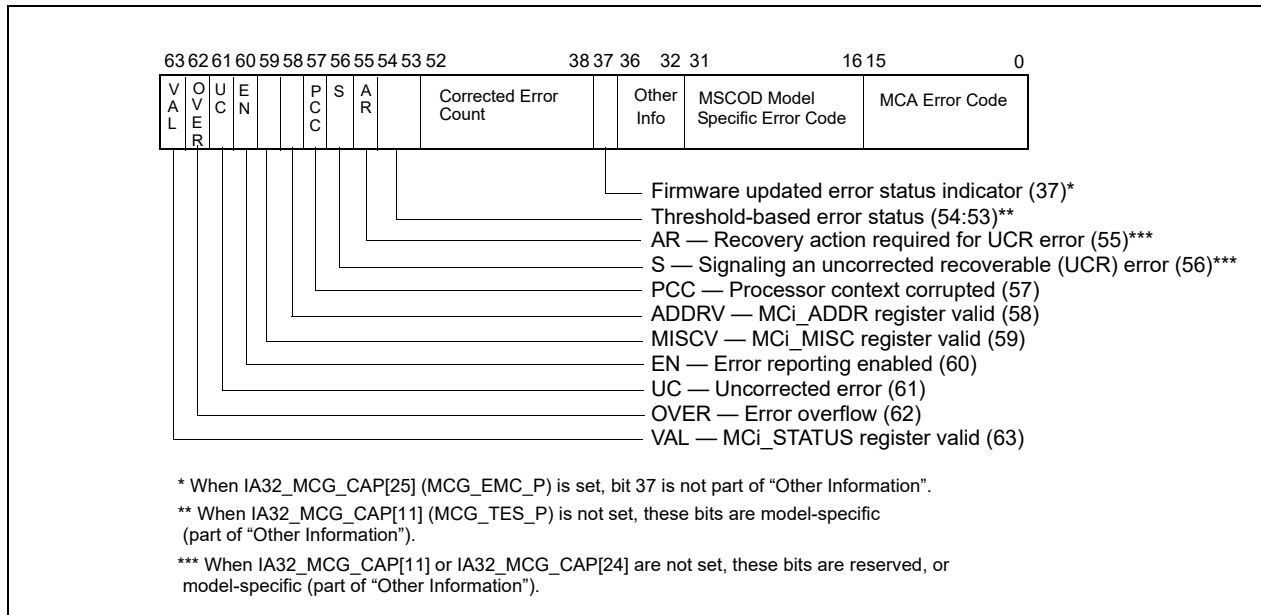


Figure 15-6. IA32_MCI_STATUS Register

- If IA32_MCG_CAP.MCG_CMCI_P[bit 10] is 0, bits 52:38 also contain "Other Information" (in the same sense as bits 37:32).
- If IA32_MCG_CAP[10] is 1, bits 52:38 are architectural (not model-specific). In this case, bits 52:38 reports the value of a 15 bit counter that increments each time a corrected error is observed by the MCA recording bank. This count value will continue to increment until cleared by software. The most significant bit, 52, is a sticky count overflow bit.
- If IA32_MCG_CAP[11] is 0, bits 56:53 also contain "Other Information" (in the same sense).
- If IA32_MCG_CAP[11] is 1, bits 56:53 are architectural (not model-specific). In this case, bits 56:53 have the following functionality:
 - If IA32_MCG_CAP[24] is 0, bits 56:55 are reserved.
 - If IA32_MCG_CAP[24] is 1, bits 56:55 are defined as follows:
 - S (Signaling) flag, bit 56 - Signals the reporting of UCR errors in this MC bank. See Section 15.6.2 for additional detail.
 - AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. See Section 15.6.2 for additional detail.
 - If the UC bit (Figure 15-6) is 1, bits 54:53 are undefined.
 - If the UC bit (Figure 15-6) is 0, bits 54:53 indicate the status of the hardware structure that reported the threshold-based error. See Table 15-1.

Table 15-1. Bits 54:53 in IA32_MCI_STATUS MSRs when IA32_MCG_CAP[11] = 1 and UC = 0

Bits 54:53	Meaning
00	No tracking - No hardware status tracking is provided for the structure reporting this event.
01	Green - Status tracking is provided for the structure posting the event; the current status is green (below threshold). For more information, see Section 15.4, "Enhanced Cache Error reporting".
10	Yellow - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). For more information, see Section 15.4, "Enhanced Cache Error reporting".
11	Reserved

- **PCC (processor context corrupt) flag, bit 57** — Indicates (when set) that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state, and software may be able to restart. When system software supports recovery, consult Section 15.10.4, “Machine-Check Software Handler Guidelines for Error Recovery” for additional rules that apply.
- **ADDRV (IA32_MCi_ADDR register valid) flag, bit 58** — Indicates (when set) that the IA32_MCi_ADDR register contains the address where the error occurred (see Section 15.3.2.3, “IA32_MCi_ADDR MSRs”). When clear, this flag indicates that the IA32_MCi_ADDR register is either not implemented or does not contain the address where the error occurred. Do not read these registers if they are not implemented in the processor.
- **MISCV (IA32_MCi_MISC register valid) flag, bit 59** — Indicates (when set) that the IA32_MCi_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32_MCi_MISC register is either not implemented or does not contain additional information regarding the error. Do not read these registers if they are not implemented in the processor.
- **EN (error enabled) flag, bit 60** — Indicates (when set) that the error was enabled by the associated EEj bit of the IA32_MCi_CTL register.
- **UC (error uncorrected) flag, bit 61** — Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.
- **OVER (machine check overflow) flag, bit 62** — Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the IA32_MCi_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. In general, enabled errors are written over disabled errors, and uncorrected errors are written over corrected errors. Uncorrected errors are not written over previous valid uncorrected errors. When MCG_CMCI_P is set, corrected errors may not set the OVER flag. Software can rely on corrected error count in IA32_MCi_Status[52:38] to determine if any additional corrected errors may have occurred. For more information, see Section 15.3.2.2.1, “Overwrite Rules for Machine Check Overflow”.
- **VAL (IA32_MCi_STATUS register valid) flag, bit 63** — Indicates (when set) that the information within the IA32_MCi_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32_MCi_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

15.3.2.2.1 Overwrite Rules for Machine Check Overflow

Table 15-2 shows the overwrite rules for how to treat a second event if the cache has already posted an event to the MC bank – that is, what to do if the valid bit for an MC bank already is set to 1. When more than one structure posts events in a given bank, these rules specify whether a new event will overwrite a previous posting or not. These rules define a priority for uncorrected (highest priority), yellow, and green/unmonitored (lowest priority) status.

In Table 15-2, the values in the two left-most columns are IA32_MCi_STATUS[54:53].

Table 15-2. Overwrite Rules for Enabled Errors

First Event	Second Event	UC bit	Color	MCA Info
00/green	00/green	0	00/green	either
00/green	yellow	0	yellow	second error
yellow	00/green	0	yellow	first error
yellow	yellow	0	yellow	either
00/green/yellow	UC	1	undefined	second
UC	00/green/yellow	1	undefined	first

If a second event overwrites a previously posted event, the information (as guarded by individual valid bits) in the MCI bank is entirely from the second event. Similarly, if a first event is retained, all of the information previously posted for that event is retained. In general, when the logged error or the recent error is a corrected error, the OVER bit (MCI_Status[62]) may be set to indicate an overflow. When MCG_CMCI_P is set in IA32_MCG_CAP, system software should consult IA32_MCi_STATUS[52:38] to determine if additional corrected errors may have

occurred. Software may re-read IA32_MCi_STATUS, IA32_MCi_ADDR and IA32_MCi_MISC appropriately to ensure data collected represent the last error logged.

After software polls a posting and clears the register, the valid bit is no longer set and therefore the meaning of the rest of the bits, including the yellow/green/00 status field in bits 54:53, is undefined. The yellow/green indication will only be posted for events associated with monitored structures – otherwise the unmonitored (00) code will be posted in IA32_MCi_STATUS[54:53].

15.3.2.3 IA32_MCi_ADDR MSRs

The IA32_MCi_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDR_V flag in the IA32_MCi_STATUS register is set (see Section 15-7, “IA32_MCi_ADDR MSR”). The IA32_MCi_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCi_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception.

The address returned is an offset into a segment, linear address, or physical address. This depends on the error encountered. When these registers are implemented, these registers can be cleared by explicitly writing 0s to these registers. Writing 1s to these registers will cause a general-protection exception. See Figure 15-7.

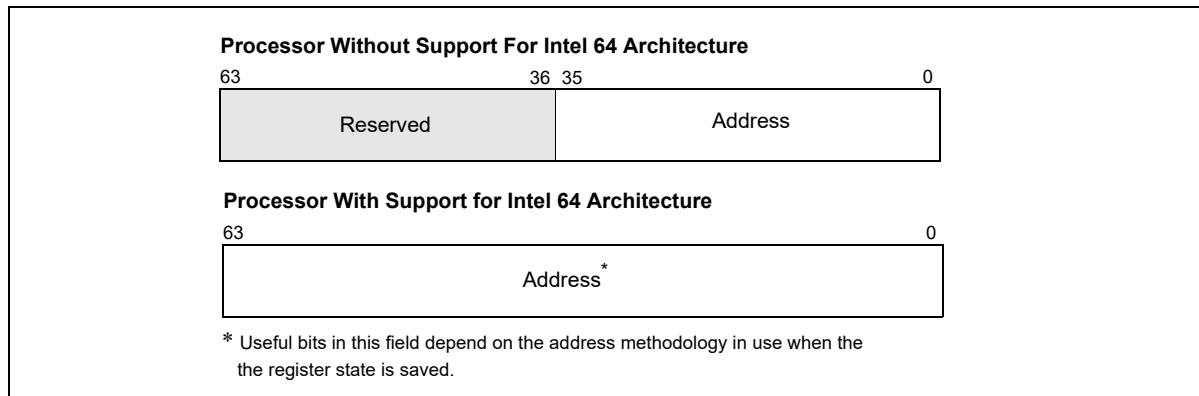


Figure 15-7. IA32_MCi_ADDR MSR

15.3.2.4 IA32_MCi_MISC MSRs

The IA32_MCi_MISC MSR contains additional information describing the machine-check error if the MISC_V flag in the IA32_MCi_STATUS register is set. The IA32_MCi_MISC_MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MCi_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception. When implemented in a processor, these registers can be cleared by explicitly writing all 0s to them; writing 1s to them causes a general-protection exception to be generated. This register is not implemented in any of the error-reporting register banks for the P6 or Intel Atom family processors.

If both MISC_V and IA32_MCG_CAP[24] are set, the IA32_MCi_MISC_MSR is defined according to Figure 15-8 to support software recovery of uncorrected errors (see Section 15.6).

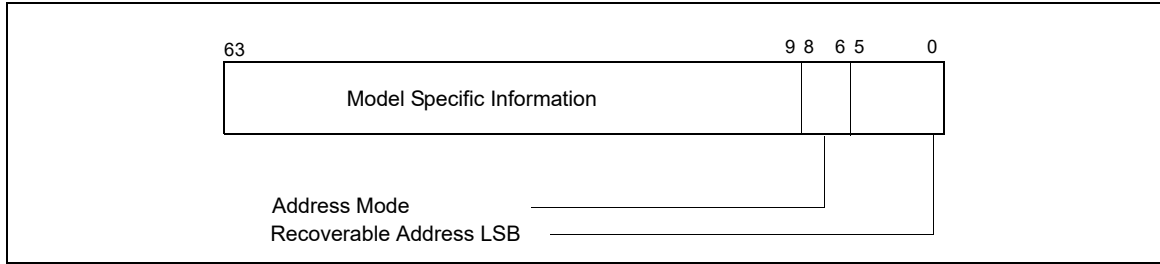


Figure 15-8. UCR Support in IA32_MCI_MISC Register

- Recoverable Address LSB (bits 5:0): The lowest valid recoverable address bit. Indicates the position of the least significant bit (LSB) of the recoverable error address. For example, if the processor logs bits [43:9] of the address, the LSB sub-field in IA32_MCI_MISC is 01001b (9 decimal). For this example, bits [8:0] of the recoverable error address in IA32_MCI_ADDR should be ignored.
- Address Mode (bits 8:6): Address mode for the address logged in IA32_MCI_ADDR. The supported address modes are given in Table 15-3.

Table 15-3. Address Mode in IA32_MCI_MISC[8:6]

IA32_MCI_MISC[8:6] Encoding	Definition
000	Segment Offset
001	Linear Address
010	Physical Address
011	Memory Address
100 to 110	Reserved
111	Generic

- Model Specific Information (bits 63:9): Not architecturally defined.

15.3.2.4.2 IOMCA

Logging and Signaling of errors from PCI Express domain is governed by PCI Express Advanced Error Reporting (AER) architecture. PCI Express architecture divides errors in two categories: Uncorrectable errors and Correctable errors. Uncorrectable errors can further be classified as Fatal or Non-Fatal. Uncorrected IO errors are signaled to the system software either as AER Message Signaled Interrupt (MSI) or via platform specific mechanisms such as NMI. Generally, the signaling mechanism is controlled by BIOS and/or platform firmware. Certain processors support an error handling mode, called IOMCA mode, where Uncorrected PCI Express errors are signaled in the form of machine check exception and logged in machine check banks.

When a processor is in this mode, Uncorrected PCI Express errors are logged in the MCACOD field of the IA32_MCI_STATUS register as Generic I/O error. The corresponding MCA error code is defined in Table 15-8. IA32_MCI_Status [15:0] Simple Error Code Encoding. Machine check logging complements and does not replace AER logging that occurs inside the PCI Express hierarchy. The PCI Express Root Complex and Endpoints continue to log the error in accordance with PCI Express AER mechanism. In IOMCA mode, MCI_MISC register in the bank that logged IOMCA can optionally contain information that link the Machine Check logs with the AER logs or proprietary logs. In such a scenario, the machine check handler can utilize the contents of MCI_MISC to locate the next level of error logs corresponding to the same error. Specifically, if MCI_Status.MISCV is 1 and MCACOD is 0x0E0B, MCI_MISC contains the PCI Express address of the Root Complex device containing the AER Logs. Software can consult the header type and class code registers in the Root Complex device's PCIe Configuration space to determine what type of device it is. This Root Complex device can either be a PCI Express Root Port, PCI Express Root Complex Event Collector or a proprietary device.

Errors that originate from PCI Express or Legacy Endpoints are logged in the corresponding Root Port in addition to the generating device. If `MISCV=1` and `MCi_MISC` contains the address of the Root Port or a Root Complex Event collector, software can parse the AER logs to learn more about the error.

If `MISCV=1` and `MCi_MISC` points to a device that is neither a Root Complex Event Collector nor a Root Port, software must consult the Vendor ID/Device ID and use device specific knowledge to locate and interpret the error log registers. In some cases, the Root Complex device configuration space may not be accessible to the software and both the Vendor and Device ID read as `0xFFFF`.

- The format of `MCi_MISC` for IOMCA errors is shown in Table 15-4.

Table 15-4. Address Mode in IA32_MCi_MISC[8:6]

63:40	39:32	31:16	15:9	8:6	5:0
RSVD	PCI Express Segment number	PCI Express Requestor ID	RSVD	ADDR MODE ¹	RECOV ADDR LSB ¹

NOTES:

- Not Applicable if `ADDRV=0`.

Refer to PCI Express Specification 3.0 for definition of PCI Express Requestor ID and AER architecture. Refer to PCI Firmware Specification 3.0 for an explanation of PCI Express Segment number and how software can access configuration space of a PCI Express device given the segment number and Requestor ID.

15.3.2.5 IA32_MCi_CTL2 MSRs

The `IA32_MCi_CTL2` MSR provides the programming interface to use corrected MC error signaling capability that is indicated by `IA32_MCG_CAP[10] = 1`. Software must check for the presence of `IA32_MCi_CTL2` on a per-bank basis.

When `IA32_MCG_CAP[10] = 1`, the `IA32_MCi_CTL2` MSR for each bank exists, i.e. reads and writes to these MSR are supported. However, signaling interface for corrected MC errors may not be supported in all banks.

The layout of `IA32_MCi_CTL2` is shown in Figure 15-9:

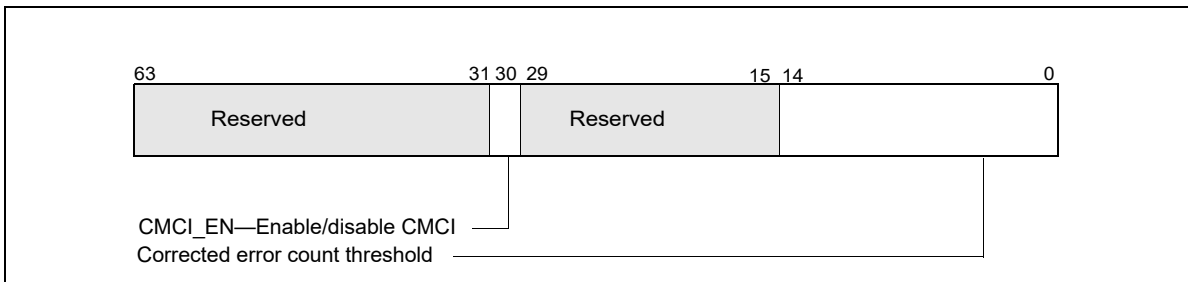


Figure 15-9. IA32_MCi_CTL2 Register

- Corrected error count threshold, bits 14:0** — Software must initialize this field. The value is compared with the corrected error count field in `IA32_MCi_STATUS`, bits 38 through 52. An overflow event is signaled to the CMCI LVT entry (see Table 10-1) in the APIC when the count value equals the threshold value. The new LVT entry in the APIC is at `02F0H` offset from the `APIC_BASE`. If CMCI interface is not supported for a particular bank (but `IA32_MCG_CAP[10] = 1`), this field will always read 0.
- CMCI_EN (Corrected error interrupt enable/disable/indicator), bits 30** — Software sets this bit to enable the generation of corrected machine-check error interrupt (CMCI). If CMCI interface is not supported for a particular bank (but `IA32_MCG_CAP[10] = 1`), this bit is writeable but will always return 0 for that bank. This bit also indicates CMCI is supported or not supported in the corresponding bank. See Section 15.5 for details of software detection of CMCI facility.

Some microarchitectural sub-systems that are the source of corrected MC errors may be shared by more than one logical processors. Consequently, the facilities for reporting MC errors and controlling mechanisms may be shared by more than one logical processors. For example, the IA32_MCi_CTL2 MSR is shared between logical processors sharing a processor core. Software is responsible to program IA32_MCi_CTL2 MSR in a consistent manner with CMCi delivery and usage.

After processor reset, IA32_MCi_CTL2 MSRs are zero'ed.

15.3.2.6 IA32_MCG Extended Machine Check State MSRs

The Pentium 4 and Intel Xeon processors implement a variable number of extended machine-check state MSRs. The MCG_EXT_P flag in the IA32_MCG_CAP MSR indicates the presence of these extended registers, and the MCG_EXT_CNT field indicates the number of these registers actually implemented. See Section 15.3.1.1, "IA32_MCG_CAP MSR." Also see Table 15-5.

Table 15-5. Extended Machine Check State MSRs in Processors Without Support for Intel 64 Architecture

MSR	Address	Description
IA32_MCG_EAX	180H	Contains state of the EAX register at the time of the machine-check error.
IA32_MCG_EBX	181H	Contains state of the EBX register at the time of the machine-check error.
IA32_MCG_ECX	182H	Contains state of the ECX register at the time of the machine-check error.
IA32_MCG_EDX	183H	Contains state of the EDX register at the time of the machine-check error.
IA32_MCG_ESI	184H	Contains state of the ESI register at the time of the machine-check error.
IA32_MCG EDI	185H	Contains state of the EDI register at the time of the machine-check error.
IA32_MCG_EBP	186H	Contains state of the EBP register at the time of the machine-check error.
IA32_MCG_ESP	187H	Contains state of the ESP register at the time of the machine-check error.
IA32_MCG_EFLAGS	188H	Contains state of the EFLAGS register at the time of the machine-check error.
IA32_MCG_EIP	189H	Contains state of the EIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

In processors with support for Intel 64 architecture, 64-bit machine check state MSRs are aliased to the legacy MSRs. In addition, there may be registers beyond IA32_MCG_MISC. These may include up to five reserved MSRs (IA32_MCG_RESERVED[1:5]) and save-state MSRs for registers introduced in 64-bit mode. See Table 15-6.

Table 15-6. Extended Machine Check State MSRs In Processors With Support For Intel 64 Architecture

MSR	Address	Description
IA32_MCG_RAX	180H	Contains state of the RAX register at the time of the machine-check error.
IA32_MCG_RBX	181H	Contains state of the RBX register at the time of the machine-check error.
IA32_MCG_RCX	182H	Contains state of the RCX register at the time of the machine-check error.
IA32_MCG_RDX	183H	Contains state of the RDX register at the time of the machine-check error.
IA32_MCG_RSI	184H	Contains state of the RSI register at the time of the machine-check error.
IA32_MCG_RDI	185H	Contains state of the RDI register at the time of the machine-check error.
IA32_MCG_RBP	186H	Contains state of the RBP register at the time of the machine-check error.
IA32_MCG_RSP	187H	Contains state of the RSP register at the time of the machine-check error.
IA32_MCG_RFLAGS	188H	Contains state of the RFLAGS register at the time of the machine-check error.
IA32_MCG_RIP	189H	Contains state of the RIP register at the time of the machine-check error.

**Table 15-6. Extended Machine Check State MSRs
In Processors With Support For Intel 64 Architecture (Contd.)**

MSR	Address	Description
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.
IA32_MCG_RSERVED[1:5]	18BH-18FH	These registers, if present, are reserved.
IA32_MCG_R8	190H	Contains state of the R8 register at the time of the machine-check error.
IA32_MCG_R9	191H	Contains state of the R9 register at the time of the machine-check error.
IA32_MCG_R10	192H	Contains state of the R10 register at the time of the machine-check error.
IA32_MCG_R11	193H	Contains state of the R11 register at the time of the machine-check error.
IA32_MCG_R12	194H	Contains state of the R12 register at the time of the machine-check error.
IA32_MCG_R13	195H	Contains state of the R13 register at the time of the machine-check error.
IA32_MCG_R14	196H	Contains state of the R14 register at the time of the machine-check error.
IA32_MCG_R15	197H	Contains state of the R15 register at the time of the machine-check error.

When a machine-check error is detected on a Pentium 4 or Intel Xeon processor, the processor saves the state of the general-purpose registers, the R/EFLAGS register, and the R/EIP in these extended machine-check state MSRs. This information can be used by a debugger to analyze the error.

These registers are read/write to zero registers. This means software can read them; but if software writes to them, only all zeros is allowed. If software attempts to write a non-zero value into one of these registers, a general-protection (#GP) exception is generated. These registers are cleared on a hardware reset (power-up or RESET), but maintain their contents following a soft reset (INIT reset).

15.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture

The Pentium processor reports machine-check errors using two registers: P5_MC_TYPE and P5_MC_ADDR. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors map these registers to the IA32_MC_i_STATUS and IA32_MC_i_ADDR in the error-reporting register bank. This bank reports on the same type of external bus errors reported in P5_MC_TYPE and P5_MC_ADDR.

The information in these registers can then be accessed in two ways:

- By reading the IA32_MC_i_STATUS and IA32_MC_i_ADDR registers as part of a general machine-check exception handler written for Pentium 4, Intel Atom and P6 family processors.
- By reading the P5_MC_TYPE and P5_MC_ADDR registers using the RDMSR instruction.

The second capability permits a machine-check exception handler written to run on a Pentium processor to be run on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor. There is a limitation in that information returned by the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors is encoded differently than information returned by the Pentium processor. To run a Pentium processor machine-check exception handler on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor; the handler must be written to interpret P5_MC_TYPE encodings correctly.

15.4 ENHANCED CACHE ERROR REPORTING

Starting with Intel Core Duo processors, cache error reporting was enhanced. In earlier Intel processors, cache status was based on the number of correction events that occurred in a cache. In the new paradigm, called "threshold-based error status", cache status is based on the number of lines (ECC blocks) in a cache that incur repeated corrections. The threshold is chosen by Intel, based on various factors. If a processor supports threshold-based error status, it sets IA32_MCG_CAP[11] (MCG_TES_P) to 1; if not, to 0.

A processor that supports enhanced cache error reporting contains hardware that tracks the operating status of certain caches and provides an indicator of their “health”. The hardware reports a “green” status when the number of lines that incur repeated corrections is at or below a pre-defined threshold, and a “yellow” status when the number of affected lines exceeds the threshold. Yellow status means that the cache reporting the event is operating correctly, but you should schedule the system for servicing within a few weeks.

Intel recommends that you rely on this mechanism for structures supported by threshold-base error reporting.

The CPU/system/platform response to a yellow event should be less severe than its response to an uncorrected error. An uncorrected error means that a serious error has actually occurred, whereas the yellow condition is a warning that the number of affected lines has exceeded the threshold but is not, in itself, a serious event: the error was corrected and system state was not compromised.

The green/yellow status indicator is not a foolproof early warning for an uncorrected error resulting from the failure of two bits in the same ECC block. Such a failure can occur and cause an uncorrected error before the yellow threshold is reached. However, the chance of an uncorrected error increases as the number of affected lines increases.

15.5 CORRECTED MACHINE CHECK ERROR INTERRUPT

Corrected machine-check error interrupt (CMCI) is an architectural enhancement to the machine-check architecture. It provides capabilities beyond those of threshold-based error reporting (Section 15.4). With threshold-based error reporting, software is limited to use periodic polling to query the status of hardware corrected MC errors. CMCI provides a signaling mechanism to deliver a local interrupt based on threshold values that software can program using the IA32_MCi_CTL2 MSRs.

CMCI is disabled by default. System software is required to enable CMCI for each IA32_MCi bank that support the reporting of hardware corrected errors if IA32_MCG_CAP[10] = 1.

System software use IA32_MCi_CTL2 MSR to enable/disable the CMCI capability for each bank and program threshold values into IA32_MCi_CTL2 MSR. CMCI is not affected by the CR4.MCE bit, and it is not affected by the IA32_MCi_CTL MSRs.

To detect the existence of thresholding for a given bank, software writes only bits 14:0 with the threshold value. If the bits persist, then thresholding is available (and CMCI is available). If the bits are all 0's, then no thresholding exists. To detect that CMCI signaling exists, software writes a 1 to bit 30 of the MCI_CTL2 register. Upon subsequent read, if bit 30 = 0, no CMCI is available for this bank and no corrected or UCNA errors will be reported on this bank. If bit 30 = 1, then CMCI is available and enabled.

15.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 15-10.

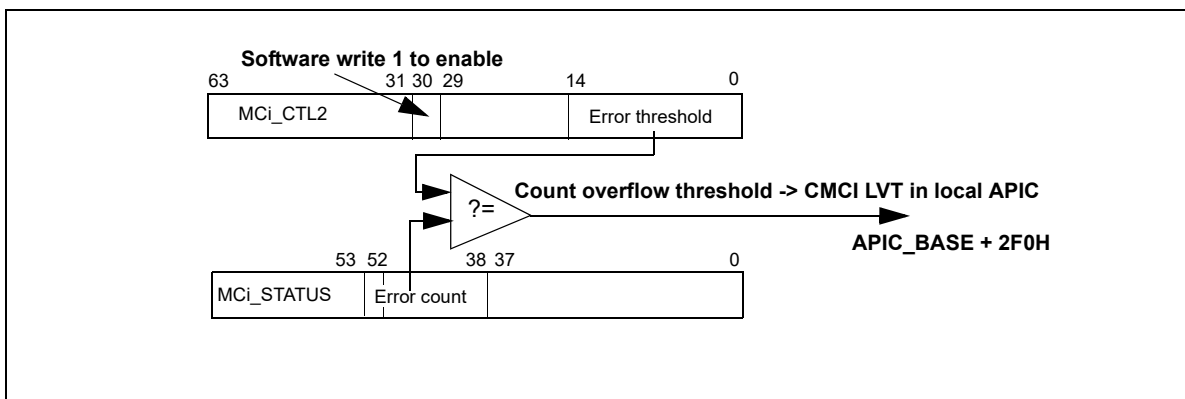


Figure 15-10. CMCI Behavior

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of `APIC_BASE + 2F0H`. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both logical processors sharing that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 10.5.1, “Local Vector Table” for details regarding the LVT CMCI register.

15.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources

System software must enable and manage CMCI, set up interrupt handlers to service CMCI interrupts delivered to affected logical processors, program CMCI LVT entry, and query machine check banks that are shared by more than one logical processors.

This section describes techniques system software can implement to manage CMCI initialization, service CMCI interrupts in an efficient manner to minimize contentions to access shared MSR resources.

15.5.2.1 CMCI Initialization

Although a CMCI interrupt may be delivered to more than one logical processors depending on the nature of the corrected MC error, only one instance of the interrupt service routine needs to perform the necessary service and make queries to the machine-check banks. The following steps describes a technique that limits the amount of work the system has to do in response to a CMCI.

- To provide maximum flexibility, system software should define per-thread data structure for each logical processor to allow equal-opportunity and efficient response to interrupt delivery. Specifically, the per-thread data structure should include a set of per-bank fields to track which machine check bank it needs to access in response to a delivered CMCI interrupt. The number of banks that needs to be tracked is determined by `IA32_MCG_CAP[7:0]`.
- Initialization of per-thread data structure. The initialization of per-thread data structure must be done serially on each logical processor in the system. The sequencing order to start the per-thread initialization between different logical processor is arbitrary. But it must observe the following specific detail to satisfy the shared nature of specific MSR resources:
 - a. Each thread initializes its data structure to indicate that it does not own any MC bank registers.
 - b. Each thread examines `IA32_MCi_CTL2[30]` indicator for each bank to determine if another thread has already claimed ownership of that bank.
 - If `IA32_MCi_CTL2[30]` had been set by another thread. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
 - If `IA32_MCi_CTL2[30] = 0`, proceed to step c.
 - c. Check whether writing a 1 into `IA32_MCi_CTL2[30]` can return with 1 on a subsequent read to determine this bank can support CMCI.
 - If `IA32_MCi_CTL2[30] = 0`, this bank does not support CMCI. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
 - If `IA32_MCi_CTL2[30] = 1`, modify the per-thread data structure to indicate this thread claims ownership to the MC bank; proceed to initialize the error threshold count (bits 15:0) of that bank as described in Chapter 15, “CMCI Threshold Management”. Then proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
- After the thread has examined all of the machine check banks, it sees if it owns any MC banks to service CMCI. If any bank has been claimed by this thread:
 - Ensure that the CMCI interrupt handler has been set up as described in Chapter 15, “CMCI Interrupt Handler”.
 - Initialize the CMCI LVT entry, as described in Section 15.5.1, “CMCI Local APIC Interface”.

- Log and clear all of IA32_MCi_Status registers for the banks that this thread owns. This will allow new errors to be logged.

15.5.2.2 CMCI Threshold Management

The Corrected MC error threshold field, IA32_MCi_CTL2[14:0], is architecturally defined. Specifically, all these bits are writable by software, but different processor implementations may choose to implement less than 15 bits as threshold for the overflow comparison with IA32_MCi_STATUS[52:38]. The following describes techniques that software can manage CMCI threshold to be compatible with changes in implementation characteristics:

- Software can set the initial threshold value to 1 by writing 1 to IA32_MCi_CTL2[14:0]. This will cause overflow condition on every corrected MC error and generates a CMCI interrupt.
- To increase the threshold and reduce the frequency of CMCI servicing:
 - a. Find the maximum threshold value a given processor implementation supports. The steps are:
 - Write 7FFFH to IA32_MCi_CTL2[14:0],
 - Read back IA32_MCi_CTL2[14:0]; these 15 bits (14:0) contain the maximum threshold supported by the processor.
 - b. Increase the threshold to a value below the maximum value discovered using step a.

15.5.2.3 CMCI Interrupt Handler

The following describes techniques system software may consider to implement a CMCI service routine:

- The service routine examines its private per-thread data structure to check which set of MC banks it has ownership. If the thread does not have ownership of a given MC bank, proceed to the next MC bank. Ownership is determined at initialization time which is described in Section 15.5.2.1.

If the thread had claimed ownership to an MC bank, this technique will allow each logical processors to handle corrected MC errors independently and requires no synchronization to access shared MSR resources. Consult Example 15-5 for guidelines on logging when processing CMCI.

15.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_2EH (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*). This allow system software to perform recovery action on certain class of uncorrected errors and continue execution.

15.6.1 Detection of Software Error Recovery Support

Software must use bit 24 of IA32_MCG_CAP (MCG_SER_P) to detect the presence of software error recovery support (see Figure 15-2). When IA32_MCG_CAP[24] is set, this indicates that the processor supports software error recovery. When this bit is clear, this indicates that there is no support for error recovery from the processor and the primary responsibility of the machine check handler is logging the machine check error information and shutting down the system.

The new class of architectural MCA errors from which system software can attempt recovery is called Uncorrected Recoverable (UCR) Errors. UCR errors are uncorrected errors that have been detected and signaled but have not corrupted the processor context. For certain UCR errors, this means that once system software has performed a certain recovery action, it is possible to continue execution on this processor. UCR error reporting provides an error containment mechanism for data poisoning. The machine check handler will use the error log information from the error reporting registers to analyze and implement specific error recovery actions for UCR errors.

15.6.2 UCR Error Reporting and Logging

IA32_MCi_STATUS MSR is used for reporting UCR errors and existing corrected or uncorrected errors. The definitions of IA32_MCi_STATUS, including bit fields to identify UCR errors, is shown in Figure 15-6. UCR errors can be signaled through either the corrected machine check interrupt (CMCI) or machine check exception (MCE) path depending on the type of the UCR error.

When IA32_MCG_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32_MCi_STATUS register:

- Valid (bit 63) = 1
- UC (bit 61) = 1
- PCC (bit 57) = 0

Additional information from the IA32_MCi_MISC and the IA32_MCi_ADDR registers for the UCR error are available when the ADDR_V and the MISC_V flags in the IA32_MCi_STATUS register are set (see Section 15.3.2.4). The MCA error code field of the IA32_MCi_STATUS register indicates the type of UCR error. System software can interpret the MCA error code field to analyze and identify the necessary recovery action for the given UCR error.

In addition, the IA32_MCi_STATUS register bit fields, bits 56:55, are defined (see Figure 15-6) to provide additional information to help system software to properly identify the necessary recovery action for the UCR error:

- S (Signaling) flag, bit 56 - Indicates (when set) that a machine check exception was generated for the UCR error reported in this MC bank and system software needs to check the AR flag and the MCA error code fields in the IA32_MCi_STATUS register to identify the necessary recovery action for this error. When the S flag in the IA32_MCi_STATUS register is clear, this UCR error was not signaled via a machine check exception and instead was reported as a corrected machine check (CMC). System software is not required to take any recovery action when the S flag in the IA32_MCi_STATUS register is clear.
- AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. This recovery action must be completed successfully before any additional work is scheduled for this processor. When the RIP_V flag in the IA32_MCG_STATUS is clear, an alternative execution stream needs to be provided; when the MCA error code specific recovery specific recovery action cannot be successfully completed, system software must shut down the system. When the AR flag in the IA32_MCi_STATUS register is clear, system software may still take MCA error code specific recovery action but this is optional; system software can safely resume program execution at the instruction pointer saved on the stack from the machine check exception when the RIP_V flag in the IA32_MCG_STATUS register is set.

Both the S and the AR flags in the IA32_MCi_STATUS register are defined to be sticky bits, which mean that once set, the processor does not clear them. Only software and good power-on reset can clear the S and the AR-flags. Both the S and the AR flags are only set when the processor reports the UCR errors (MCG_CAP[24] is set).

15.6.3 UCR Error Classification

With the S and AR flag encoding in the IA32_MCi_STATUS register, UCR errors can be classified as:

- Uncorrected no action required (UCNA) - is a UCR error that is not signaled via a machine check exception and, instead, is reported to system software as a corrected machine check error. UCNA errors indicate that some data in the system is corrupted, but the data has not been consumed and the processor state is valid and you may continue execution on this processor. UCNA errors require no action from system software to continue execution. A UCNA error is indicated with UC=1, PCC=0, S=0 and AR=0 in the IA32_MCi_STATUS register.
- Software recoverable action optional (SRAO) - a UCR error is signaled either via a machine check exception or CMCI. System software recovery action is optional and not required to continue execution from this machine check exception. SRAO errors indicate that some data in the system is corrupt, but the data has not been consumed and the processor state is valid. SRAO errors provide the additional error information for system software to perform a recovery action. An SRAO error when signaled as a machine check is indicated with UC=1, PCC=0, S=1, EN=1 and AR=0 in the IA32_MCi_STATUS register. In cases when SRAO is signaled via CMCI the error signature is indicated via UC=1, PCC=0, S=0. Recovery actions for SRAO errors are MCA error code specific. The MISC_V and the ADDR_V flags in the IA32_MCi_STATUS register are set when the additional error information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCi_STATUS register to identify the specific recovery

action for a given SRAO error. If MISCV and ADDRIV are not set, it is recommended that no system software error recovery be performed however, system software can resume execution.

- Software recoverable action required (SRAR) - a UCR error that requires system software to take a recovery action on this processor before scheduling another stream of execution on this processor. SRAR errors indicate that the error was detected and raised at the point of the consumption in the execution flow. An SRAR error is indicated with UC=1, PCC=0, S=1, EN=1 and AR=1 in the IA32_MCi_STATUS register. Recovery actions are MCA error code specific. The MISCV and the ADDRIV flags in the IA32_MCi_STATUS register are set when the additional error information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCi_STATUS register to identify the specific recovery action for a given SRAR error. If MISCV and ADDRIV are not set, it is recommended that system software shutdown the system.

Table 15-7 summarizes UCR, corrected, and uncorrected errors.

Table 15-7. MC Error Classifications

Type of Error ¹	UC	EN	PCC	S	AR	Signaling	Software Action	Example
Uncorrected Error (UC)	1	1	1	x	x	MCE	If EN=1, reset the system, else log and OK to keep the system running.	
SRAR	1	1	0	1	1	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck. If OVER=1, reset system, else take specific recovery action.	Cache to processor load error.
SRAO	1	x ²	0	x ²	0	MCE/CMC	For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running.	Patrol scrub and explicit writeback poison errors.
UCNA	1	x	0	0	0	CMC	Log the error and Ok to keep the system running.	Poison detection error.
Corrected Error (CE)	0	x	x	x	x	CMC	Log the error and no corrective action required.	ECC in caches and memory.

NOTES:

1. SRAR, SRAO and UCNA errors are supported by the processor only when IA32_MCG_CAP[24] (MCG_SER_P) is set.
2. EN=1, S=1 when signaled via MCE. EN=x, S=0 when signaled via CMC.

15.6.4 UCR Error Overwrite Rules

In general, the overwrite rules are as follows:

- UCR errors will overwrite corrected errors.
- Uncorrected (PCC=1) errors overwrite UCR (PCC=0) errors.
- UCR errors are not written over previous UCR errors.
- Corrected errors do not write over previous UCR errors.

Regardless of whether the 1st error is retained or the 2nd error is overwritten over the 1st error, the OVER flag in the IA32_MCi_STATUS register will be set to indicate an overflow condition. As the S flag and AR flag in the IA32_MCi_STATUS register are defined to be sticky flags, a second event cannot clear these 2 flags once set, however the MC bank information may be filled in for the 2nd error. The table below shows the overwrite rules and how to treat a second error if the first event is already logged in a MC bank along with the resulting bit setting of the UC, PCC, and AR flags in the IA32_MCi_STATUS register. As UCNA and SRAO errors do not require recovery action from system software to continue program execution, a system reset by system software is not required unless the AR flag or PCC flag is set for the UCR overflow case (OVER=1, VAL=1, UC=1, PCC=0).

Table 15-8 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

Table 15-8. Overwrite Rules for UC, CE, and UCR Errors

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
CE	UCR	1	0	0 if UCNA, else 1	1 if SRAR, else 0	second	yes, if AR=1
UCR	CE	1	0	0 if UCNA, else 1	1 if SRAR, else 0	first	yes, if AR=1
UCNA	UCNA	1	0	0	0	first	no
UCNA	SRAO	1	0	1	0	first	no
UCNA	SRAR	1	0	1	1	first	yes
SRAO	UCNA	1	0	1	0	first	no
SRAO	SRAO	1	0	1	0	first	no
SRAO	SRAR	1	0	1	1	first	yes
SRAR	UCNA	1	0	1	1	first	yes
SRAR	SRAO	1	0	1	1	first	yes
SRAR	SRAR	1	0	1	1	first	yes
UCR	UC	1	1	undefined	undefined	second	yes
UC	UCR	1	1	undefined	undefined	first	yes

15.7 MACHINE-CHECK AVAILABILITY

The machine-check architecture and machine-check exception (#MC) are model-specific features. Software can execute the CPUID instruction to determine whether a processor implements these features. Following the execution of the CPUID instruction, the settings of the MCA flag (bit 14) and MCE flag (bit 7) in EDX indicate whether the processor implements the machine-check architecture and machine-check exception.

15.8 MACHINE-CHECK INITIALIZATION

To use the processors machine-check architecture, software must initialize the processor to activate the machine-check exception and the error-reporting mechanism.

Example 15-1 gives pseudocode for performing this initialization. This pseudocode checks for the existence of the machine-check architecture and exception; it then enables machine-check exception and the error-reporting register banks. The pseudocode shown is compatible with the Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors.

Following power up or power cycling, IA32_MCi_STATUS registers are not guaranteed to have valid data until after they are initially cleared to zero by software (as shown in the initialization pseudocode in Example 15-1).

Example 15-1. Machine-Check Initialization Pseudocode

Check CPUID Feature Flags for MCE and MCA support

IF CPU supports MCE

THEN

IF CPU supports MCA

THEN

IF (IA32_MCG_CAP.MCG_CTL_P = 1)

(* IA32_MCG_CTL register is present *)

THEN

IA32_MCG_CTL ← FFFFFFFFFFFFFFFFH;

(* enables all MCA features *)

FI

IF (IA32_MCG_CAP.MCG_LMCE_P = 1 and IA32_FEATURE_CONTROL.LOCK = 1 and IA32_FEATURE_CONTROL.LMCE_ENABLED = 1)

(* IA32_MCG_EXT_CTL register is present and platform has enabled LMCE to permit system software to use LMCE *)

```

THEN
  IA32_MCG_EXT_CTL ← IA32_MCG_EXT_CTL | 01H;
  (* System software enables LMCE capability for hardware to signal MCE to a single logical processor*)
FI

```

```

(* Determine number of error-reporting banks supported *)
COUNT ← IA32_MCG_CAP.Count;
MAX_BANK_NUMBER ← COUNT - 1;

```

```

IF (Processor Family is 6H and Processor EXTMODEL:MODEL is less than 1AH)
THEN
  (* Enable logging of all errors except for MCO_CTL register *)
  FOR error-reporting banks (1 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFH;
  OD

```

```

ELSE
  (* Enable logging of all errors including MCO_CTL register *)
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFH;
  OD
FI

```

```

(* BIOS clears all errors only on power-on reset *)
IF (BIOS detects Power-on reset)
THEN
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    IA32_MCi_STATUS ← 0;
  OD
ELSE
  FOR error-reporting banks (0 through MAX_BANK_NUMBER)
  DO
    (Optional for BIOS and OS) Log valid errors
    (OS only) IA32_MCi_STATUS ← 0;
  OD

```

```

FI
FI

```

Setup the Machine Check Exception (#MC) handler for vector 18 in IDT

Set the MCE bit (bit 6) in CR4 register to enable Machine-Check Exceptions

```

FI

```

15.9 INTERPRETING THE MCA ERROR CODES

When the processor detects a machine-check error condition, it writes a 16-bit error code to the MCA error code field of one of the IA32_MCi_STATUS registers and sets the VAL (valid) flag in that register. The processor may also write a 16-bit model-specific error code in the IA32_MCi_STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for Intel 64 and IA-32 processors. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32_MCi_STATUS register. If the flag is set, the machine check-exception handler must then read the MCA error code field of the register. It is the encoding of the MCA error code field [15:0] that determines the type of error being reported and not the register bank reporting it.

There are two types of MCA error codes: simple error codes and compound error codes.

15.9.1 Simple Error Codes

Table 15-9 shows the simple error codes. These unique codes indicate global error information.

Table 15-9. IA32_MCi_Status [15:0] Simple Error Code Encoding

Error Code	Binary Encoding	Meaning
No Error	0000 0000 0000 0000	No error has been reported to this bank of error-reporting registers.
Unclassified	0000 0000 0000 0001	This error has not been classified into the MCA error classes.
Microcode ROM Parity Error	0000 0000 0000 0010	Parity error in internal microcode ROM
External Error	0000 0000 0000 0011	The BINIT# from another processor caused this processor to enter machine check. ¹
FRC Error	0000 0000 0000 0100	FRC (functional redundancy check) master/slave error
Internal Parity Error	0000 0000 0000 0101	Internal parity error.
SMM Handler Code Access Violation	0000 0000 0000 0110	An attempt was made by the SMM Handler to execute outside the ranges specified by SMRR.
Internal Timer Error	0000 0100 0000 0000	Internal timer error.
I/O Error	0000 1110 0000 1011	generic I/O error.
Internal Unclassified	0000 01xx xxxx xxxx	Internal unclassified errors. ²

NOTES:

1. BINIT# assertion will cause a machine check exception if the processor (or any processor on the same external bus) has BINIT# observation enabled during power-on configuration (hardware strapping) and if machine check exceptions are enabled (by setting CR4.MCE = 1).
2. At least one X must equal one. Internal unclassified errors have not been classified.

15.9.2 Compound Error Codes

Compound error codes describe errors related to the TLBs, memory, caches, bus and interconnect logic, and internal timer. A set of sub-fields is common to all of compound errors. These sub-fields describe the type of access, level in the cache hierarchy, and type of request. Table 15-10 shows the general form of the compound error codes.

Table 15-10. IA32_MCi_Status [15:0] Compound Error Code Encoding

Type	Form	Interpretation
Generic Cache Hierarchy	000F 0000 0000 11LL	Generic cache hierarchy error
TLB Errors	000F 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Controller Errors	000F 0000 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Cache Hierarchy Errors	000F 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
Extended Memory Errors	000F 0010 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Bus and Interconnect Errors	000F 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_{T}_ERR

The “Interpretation” column in the table indicates the name of a compound error. The name is constructed by substituting mnemonics for the sub-field names given within curly braces. For example, the error code ICACHEL1_RD_ERR is constructed from the form:

{TT}CACHE{LL}_{RRRR}_ERR,

where {TT} is replaced by I, {LL} is replaced by L1, and {RRRR} is replaced by RD.

For more information on the “Form” and “Interpretation” columns, see Sections Section 15.9.2.1, “Correction Report Filtering (F) Bit” through Section 15.9.2.5, “Bus and Interconnect Errors”.

15.9.2.1 Correction Report Filtering (F) Bit

Starting with Intel Core Duo processors, bit 12 in the “Form” column in Table 15-10 is used to indicate that a particular posting to a log may be the last posting for corrections in that line/entry, at least for some time:

- 0 in bit 12 indicates “normal” filtering (original P6/Pentium4/Atom/Xeon processor meaning).
- 1 in bit 12 indicates “corrected” filtering (filtering is activated for the line/entry in the posting). Filtering means that some or all of the subsequent corrections to this entry (in this structure) will not be posted. The enhanced error reporting introduced with the Intel Core Duo processors is based on tracking the lines affected by repeated corrections (see Section 15.4, “Enhanced Cache Error reporting”). This capability is indicated by IA32_MCG_CAP[11]. Only the first few correction events for a line are posted; subsequent redundant correction events to the same line are not posted. Uncorrected events are always posted.

The behavior of error filtering after crossing the yellow threshold is model-specific. Filtering has meaning only for corrected errors (UC=0 in IA32_MCi_STATUS MSR). System software must ignore filtering bit (12) for uncorrected errors.

15.9.2.2 Transaction Type (TT) Sub-Field

The 2-bit TT sub-field (Table 15-11) indicates the type of transaction (data, instruction, or generic). The sub-field applies to the TLB, cache, and interconnect error conditions. Note that interconnect error conditions are primarily associated with P6 family and Pentium processors, which utilize an external APIC bus separate from the system bus. The generic type is reported when the processor cannot determine the transaction type.

Table 15-11. Encoding for TT (Transaction Type) Sub-Field

Transaction Type	Mnemonic	Binary Encoding
Instruction	I	00
Data	D	01
Generic	G	10

15.9.2.3 Level (LL) Sub-Field

The 2-bit LL sub-field (see Table 15-12) indicates the level in the memory hierarchy where the error occurred (level 0, level 1, level 2, or generic). The LL sub-field also applies to the TLB, cache, and interconnect error conditions. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support two levels in the cache hierarchy and one level in the TLBs. Again, the generic type is reported when the processor cannot determine the hierarchy level.

Table 15-12. Level Encoding for LL (Memory Hierarchy Level) Sub-Field

Hierarchy Level	Mnemonic	Binary Encoding
Level 0	L0	00
Level 1	L1	01
Level 2	L2	10
Generic	LG	11

15.9.2.4 Request (RRRR) Sub-Field

The 4-bit RRRR sub-field (see Table 15-13) indicates the type of action associated with the error. Actions include read and write operations, prefetches, cache evictions, and snoops. Generic error is returned when the type of error cannot be determined. Generic read and generic write are returned when the processor cannot determine the type of instruction or data request that caused the error. Eviction and snoop requests apply only to the caches. All of the other requests apply to TLBs, caches and interconnects.

Table 15-13. Encoding of Request (RRRR) Sub-Field

Request Type	Mnemonic	Binary Encoding
Generic Error	ERR	0000
Generic Read	RD	0001
Generic Write	WR	0010
Data Read	DRD	0011
Data Write	DWR	0100
Instruction Fetch	IRD	0101
Prefetch	PREFETCH	0110
Eviction	EVICT	0111
Snoop	SNOOP	1000

15.9.2.5 Bus and Interconnect Errors

The bus and interconnect errors are defined with the 2-bit PP (participation), 1-bit T (time-out), and 2-bit II (memory or I/O) sub-fields, in addition to the LL and RRRR sub-fields (see Table 15-14). The bus error conditions are implementation dependent and related to the type of bus implemented by the processor. Likewise, the interconnect error conditions are predicated on a specific implementation-dependent interconnect model that describes the connections between the different levels of the storage hierarchy. The type of bus is implementation dependent, and as such is not specified in this document. A bus or interconnect transaction consists of a request involving an address and a response.

Table 15-14. Encodings of PP, T, and II Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
PP (Participation)	Local processor* originated request	SRC	00
	Local processor* responded to request	RES	01
	Local processor* observed error as third party	OBS	10
	Generic		11
T (Time-out)	Request timed out	TIMEOUT	1
	Request did not time out	NOTIMEOUT	0
II (Memory or I/O)	Memory Access	M	00
	Reserved		01
	I/O	IO	10
	Other transaction		11

NOTE:

* Local processor differentiates the processor reporting the error from other system components (including the APIC, other processors, etc.).

15.9.2.6 Memory Controller and Extended Memory Errors

The memory controller errors are defined with the 3-bit MMM (memory transaction type), and 4-bit CCCC (channel) sub-fields. The encodings for MMM and CCCC are defined in Table 15-15. Extended Memory errors use the same encodings and are used to report errors in memory used as a cache.

Table 15-15. Encodings of MMM and CCCC Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
MMM	Generic undefined request	GEN	000
	Memory read error	RD	001
	Memory write error	WR	010
	Address/Command Error	AC	011
	Memory Scrubbing Error	MS	100
	Reserved		101-111
CCCC	Channel number	CHN	0000-1110
	Channel not specified		1111

Note that the CCCC channel number may be enumerated from zero separately by each memory controller on a system. On a multi-socket system, or a system with multiple memory controllers per socket, it is necessary to also consider which machine check bank logged the error. See Chapter 16 for details on specific implementations.

15.9.3 Architecturally Defined UCR Errors

Software recoverable compound error code are defined in this section.

15.9.3.1 Architecturally Defined SRAO Errors

The following two SRAO errors are architecturally defined.

- UCR Errors detected by memory controller scrubbing; and
- UCR Errors detected during L3 cache (L3) explicit writebacks.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 15-10). Their values and compound encoding format are given in Table 15-16.

Table 15-16. MCA Compound Error Code Encoding for SRAO Errors

Type	MCACOD Value	MCA Error Code Encoding ¹
Memory Scrubbing	COH - CFH	0000_0000_1100_CCCC 000F 0000 1MMM CCCC (Memory Controller Error), where Memory subfield MMM = 100B (memory scrubbing) Channel subfield CCCC = channel # or generic
L3 Explicit Writeback	17AH	0000_0001_0111_1010 000F 0001 RRRR TTLL (Cache Hierarchy Error) where Request subfields RRRR = 0111B (Eviction) Transaction Type subfields TT = 10B (Generic) Level subfields LL = 10B

NOTES:

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 15-17 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAO errors.

Table 15-17. IA32_MCi_STATUS Values for SRAO Errors

SRAO Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Memory Scrubbing	1	0	1	x ¹	1	1	0	x ¹	0	COH-CFH
L3 Explicit Writeback	1	0	1	x ¹	1	1	0	x ¹	0	17AH

NOTES:

1. When signaled as MCE, EN=1 and S=1. If error was signaled via CMC, then EN=x, and S=0.

For both the memory scrubbing and L3 explicit writeback errors, the ADDRv and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors as outlined in Section 15.10.4.1. If LMCE is supported and enabled, some errors (not limited to UCR errors) may be delivered to only a single logical processor. System software should consult IA32_MCG_STATUS.LMCE_S to determine if the MCE signaled is only to this logical processor.

IA32_MCi_STATUS banks can be shared by logical processors within a core or within the same package. So several logical processors may find an SRAO error in the shared IA32_MCi_STATUS bank but other processors do not find it in any of the IA32_MCi_STATUS banks. Table 15-18 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the memory scrubbing and L3 explicit writeback errors on both the reporting and non-reporting logical processors.

Table 15-18. IA32_MCG_STATUS Flag Indication for SRAO Errors

SRAO Type	Reporting Logical Processors		Non-reporting Logical Processors	
	RIPV	EIPV	RIPV	EIPV
Memory Scrubbing	1	0	1	0
L3 Explicit Writeback	1	0	1	0

15.9.3.2 Architecturally Defined SRAR Errors

The following two SRAR errors are architecturally defined.

- UCR Errors detected on data load; and
- UCR Errors detected on instruction fetch.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 15-10). Their values and compound encoding format are given in Table 15-19.

Table 15-19. MCA Compound Error Code Encoding for SRAR Errors

Type	MCACOD Value	MCA Error Code Encoding ¹
Data Load	134H	0000_0001_0011_0100 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0011B (Data Load) Transaction Type subfield TT= 01B (Data) Level subfield LL = 00B (Level 0)
Instruction Fetch	150H	0000_0001_0101_0000 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0101B (Instruction Fetch) Transaction Type subfield TT= 00B (Instruction) Level subfield LL = 00B (Level 0)

NOTES:

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 15-20 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAR errors.

Table 15-20. IA32_MCi_STATUS Values for SRAR Errors

SRAR Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Data Load	1	0	1	1	1	1	0	1	1	134H
Instruction Fetch	1	0	1	1	1	1	0	1	1	150H

For both the data load and instruction fetch errors, the ADDRv and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors on the system on which the UCR errors are supported, except when the processor supports LMCE and LMCE is enabled by system software (see Section 15.3.1.5). The IA32_MCG_STATUS MSR allows system software to distinguish the affected logical processor of an SRAR error amongst logical processors that observed SRAR via MCI_STATUS bank.

Table 15-21 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the data load and instruction fetch errors on both the reporting and non-reporting logical processors. The recoverable SRAR error reported by a processor may be continuable, where the system software can interpret the context of continuable as follows: the error was isolated, contained. If software can rectify the error condition in the current instruction stream, the execution context on that logical processor can be continued without loss of information.

Table 15-21. IA32_MCG_STATUS Flag Indication for SRAR Errors

SRAR Type	Affected Logical Processor			Non-Affected Logical Processors		
	RIPV	EIPV	Continuable	RIPV	EIPV	Continuable
Recoverable-continuable	1	1	Yes ¹	1	0	Yes
Recoverable-not-continuable	0	x	No			

NOTES:

1. see the definition of the context of “continuable” above and additional detail below.

SRAR Error And Affected Logical Processors

The affected logical processor is the one that has detected and raised an SRAR error at the point of the consumption in the execution flow. The affected logical processor should find the Data Load or the Instruction Fetch error information in the IA32_MCi_STATUS register that is reporting the SRAR error.

Table 15-21 list the actionable scenarios that system software can respond to an SRAR error on an affected logical processor according to RIPV and EIPV values:

- Recoverable-Continuable SRAR Error (RIPV=1, EIPV=1):
 For Recoverable-Continuable SRAR errors, the affected logical processor should find that both the IA32_MCG_STATUS.RIPV and the IA32_MCG_STATUS.EIPV flags are set, indicating that system software may be able to restart execution from the interrupted context if it is able to rectify the error condition. If system software cannot rectify the error condition then it must treat the error as a recoverable error where restarting execution with the interrupted context is not possible. Restarting without rectifying the error condition will result in most cases with another SRAR error on the same instruction.

- Recoverable-not-continuable SRAR Error (RIPV=0, EIPV=x):

For Recoverable-not-continuable errors, the affected logical processor should find that either

- IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=1, or
- IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=0.

In either case, this indicates that the error is detected at the instruction pointer saved on the stack for this machine check exception and restarting execution with the interrupted context is not possible. System software may take the following recovery actions for the affected logical processor:

- The current executing thread cannot be continued. System software must terminate the interrupted stream of execution and provide a new stream of execution on return from the machine check handler for the affected logical processor.

SRAR Error And Non-Affected Logical Processors

The logical processors that observed but not affected by an SRAR error should find that the RIPV flag in the IA32_MCG_STATUS register is set and the EIPV flag in the IA32_MCG_STATUS register is cleared, indicating that it is safe to restart the execution at the instruction saved on the stack for the machine check exception on these processors after the recovery action is successfully taken by system software.

15.9.4 Multiple MCA Errors

When multiple MCA errors are detected within a certain detection window, the processor may aggregate the reporting of these errors together as a single event, i.e. a single machine exception condition. If this occurs, system software may find multiple MCA errors logged in different MC banks on one logical processor or find multiple MCA errors logged across different processors for a single machine check broadcast event. In order to handle multiple UCR errors reported from a single machine check event and possibly recover from multiple errors, system software may consider the following:

- Whether it can recover from multiple errors is determined by the most severe error reported on the system. If the most severe error is found to be an unrecoverable error (VAL=1, UC=1, PCC=1 and EN=1) after system software examines the MC banks of all processors to which the MCA signal is broadcast, recovery from the multiple errors is not possible and system software needs to reset the system.
- When multiple recoverable errors are reported and no other fatal condition (e.g. overflowed condition for SRAR error) is found for the reported recoverable errors, it is possible for system software to recover from the multiple recoverable errors by taking necessary recovery action for each individual recoverable error. However, system software can no longer expect one to one relationship with the error information recorded in the IA32_MCi_STATUS register and the states of the RIPV and EIPV flags in the IA32_MCG_STATUS register as the states of the RIPV and the EIPV flags in the IA32_MCG_STATUS register may indicate the information for the most severe error recorded on the processor. System software is required to use the RIPV flag indication in the IA32_MCG_STATUS register to make a final decision of recoverability of the errors and find the restart-ability requirement after examining each IA32_MCi_STATUS register error information in the MC banks.

In certain cases where system software observes more than one SRAR error logged for a single logical processor, it can no longer rely on affected threads as specified in Table 15-20 above. System software is recommended to reset the system if this condition is observed.

15.9.5 Machine-Check Error Codes Interpretation

Chapter 16, "Interpreting Machine-Check Error Codes," provides information on interpreting the MCA error code, model-specific error code, and other information error code fields. For P6 family processors, information has been included on decoding external bus errors. For Pentium 4 and Intel Xeon processors; information is included on external bus, internal timer and cache hierarchy errors.

15.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE

The machine-check architecture and error logging can be used in three different ways:

- To detect machine errors during normal instruction execution, using the machine-check exception (#MC).
- To periodically check and log machine errors.
- To examine recoverable UCR errors, determine software recoverability and perform recovery actions via a machine-check exception handler or a corrected machine-check interrupt handler.

To use the machine-check exception, the operating system or executive software must provide a machine-check exception handler. This handler may need to be designed specifically for each family of processors.

A special program or utility is required to log machine errors.

Guidelines for writing a machine-check exception handler or a machine-error logging utility are given in the following sections.

15.10.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When IA32_MCG_CAP[24] is clear, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.
- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MCi_STATUS register should be checked. See Section 15.9, "Interpreting the MCA Error Codes," for information that can be used to write an algorithm to interpret this field.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MCi_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV, PCC, and OVER flags in each IA32_MCi_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture

does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

Example 15-2 gives typical steps carried out by a machine-check exception handler.

Example 15-2. Machine-Check Exception Handler Pseudocode

```

IF CPU supports MCE
  THEN
    IF CPU supports MCA
      THEN
        call errorlogging routine; (* returns restartability *)
      FI;
    ELSE (* Pentium(R) processor compatible *)
      READ P5_MC_ADDR
      READ P5_MC_TYPE;
      report RESTARTABILITY to console;
    FI;
  IF error is not restartable
    THEN
      report RESTARTABILITY to console;
      abort system;
    FI;
  CLEAR MCIP flag in IA32_MCG_STATUS;

```

15.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family, Intel Atom and later processor families, should follow the guidelines described in Section 15.10.1 and Example 15-2 that check the processor's support of MCA.

NOTE

On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5_MC_TYPE and P5_MC_ADDR registers may produce invalid data.

When machine-check exceptions are enabled for the Pentium processor (MCE flag is set in control register CR4), the machine-check exception handler uses the RDMSR instruction to read the error type from the P5_MC_TYPE register and the machine check address from the P5_MC_ADDR register. The handler then normally reports these register values to the system console before aborting execution (see Example 15-2).

15.10.3 Logging Correctable Machine-Check Errors

The error handling routine for servicing the machine-check exceptions is responsible for logging uncorrected errors.

If a machine-check error is correctable, the processor does not generate a machine-check exception for it. To detect correctable machine-check errors, a utility program must be written that reads each of the machine-check error-reporting register banks and logs the results in an accounting file or data structure. This utility can be implemented in either of the following ways.

- A system daemon that polls the register banks on an infrequent basis, such as hourly or daily.
- A user-initiated application that polls the register banks and records the exceptions. Here, the actual polling service is provided by an operating-system driver or through the system call interface.
- An interrupt service routine servicing CMCI can read the MC banks and log the error. Please refer to Section 15.10.4.2 for guidelines on logging correctable machine checks.

Example 15-3 gives pseudocode for an error logging utility.

Example 15-3. Machine-Check Error Logging Pseudocode

```

Assume that execution is restartable;
IF the processor supports MCA
  THEN
    FOR each bank of machine-check registers
      DO
        READ IA32_MCi_STATUS;
        IF VAL flag in IA32_MCi_STATUS = 1
          THEN
            IF ADDR_V flag in IA32_MCi_STATUS = 1
              THEN READ IA32_MCi_ADDR;
            FI;
            IF MISC_V flag in IA32_MCi_STATUS = 1
              THEN READ IA32_MCi_MISC;
            FI;
            IF MCIP flag in IA32_MCG_STATUS = 1
              (* Machine-check exception is in progress *)
              AND PCC flag in IA32_MCi_STATUS = 1
              OR RIPV flag in IA32_MCG_STATUS = 0
              (* execution is not restartable *)
              THEN
                RESTARTABILITY = FALSE;
                return RESTARTABILITY to calling procedure;
            FI;
            Save time-stamp counter and processor ID;
            Set IA32_MCi_STATUS to all 0s;
            Execute serializing instruction (i.e., CPUID);
          FI;
      OD;
    FI;
  FI;

```

If the processor supports the machine-check architecture, the utility reads through the banks of error-reporting registers looking for valid register entries. It then saves the values of the IA32_MCi_STATUS, IA32_MCi_ADDR, IA32_MCi_MISC and IA32_MCG_STATUS registers for each bank that is valid. The routine minimizes processing time by recording the raw data into a system data structure or file, reducing the overhead associated with polling. User utilities analyze the collected data in an off-line environment.

When the MCIP flag is set in the IA32_MCG_STATUS register, a machine-check exception is in progress and the machine-check exception handler has called the exception logging routine.

Once the logging process has been completed the exception-handling routine must determine whether execution can be restarted, which is usually possible when damage has not occurred (The PCC flag is clear, in the IA32_MCi_STATUS register) and when the processor can guarantee that execution is restartable (the RIPV flag is set in the IA32_MCG_STATUS register). If execution cannot be restarted, the system is not recoverable and the exception-handling routine should signal the console appropriately before returning the error status to the Operating System kernel for subsequent shutdown.

The machine-check architecture allows buffering of exceptions from a given error-reporting bank although the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors do not implement this feature. The error logging routine should provide compatibility with future processors by reading each hardware error-reporting bank's IA32_MCi_STATUS register and then writing 0s to clear the OVER and VAL flags in this register. The error logging utility should re-read the IA32_MCi_STATUS register for the bank ensuring that the valid bit is clear. The processor will write the next error into the register bank and set the VAL flags.

Additional information that should be stored by the exception-logging routine includes the processor's time-stamp counter value, which provides a mechanism to indicate the frequency of exceptions. A multiprocessing operating system stores the identity of the processor node incurring the exception using a unique identifier, such as the processor's APIC ID (see Section 10.8, "Handling Interrupts").

The basic algorithm given in Example 15-3 can be modified to provide more robust recovery techniques. For example, software has the flexibility to attempt recovery using information unavailable to the hardware. Specifically, the machine-check exception handler can, after logging carefully analyze the error-reporting registers when the error-logging routine reports an error that does not allow execution to be restarted. These recovery techniques

can use external bus related model-specific information provided with the error report to localize the source of the error within the system and determine the appropriate recovery strategy.

15.10.4 Machine-Check Software Handler Guidelines for Error Recovery

15.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32_MCG_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.
- When IA32_MCG_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.
- For processors on which CPUID reports DisplayFamily_DisplayModel as 06H_0EH and onward, an MCA signal is broadcast to all logical processors in the system (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*). Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging and clearing the information in the machine check registers.
 - On processors that indicate ability for local machine-check exception (MCG_LMCE_P), hardware can choose to report the error to only a single logical processor if system software has enabled LMCE by setting IA32_MCG_EXT_CTL[LMCE_EN] = 1 as outlined in Section 15.3.1.5.
- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.
- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.
- For uncorrectable errors, the EIPV flag in the IA32_MCG_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32_MCG_STATUS register is set to 1. If it is not set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

When IA32_MCG_CAP [24] is 1, the following rules can apply when writing a machine check exception (MCE) handler to support software recovery:

- The PCC flag in each IA32_MCi_STATUS register indicates whether recovery from the error is possible for uncorrected errors (UC=1). If the PCC flag is set for enabled uncorrected errors (UC=1 and EN=1), recovery is not possible. When recovery is not possible, the MCE handler typically records the error information and signals the operating system to reset the system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether restarting the program execution from the instruction pointer saved on the stack for the machine check exception is possible. When the RIPV is set, program execution can be restarted reliably when recovery is possible. If the RIPV flag is not set, program execution cannot be restarted reliably. In this case the recovery algorithm may involve terminating the current program execution and resuming an alternate thread of execution upon return from the machine check handler when recovery is possible. When recovery is not possible, the MCE handler signals the operating system to reset the system.

- When the EN flag is zero but the VAL and UC flags are one in the IA32_MCi_STATUS register, the reported uncorrected error in this bank is not enabled. As uncorrected errors with the EN flag = 0 are not the source of machine check exceptions, the MCE handler should log and clear non-enabled errors when the S bit is set and should continue searching for enabled errors from the other IA32_MCi_STATUS registers. Note that when IA32_MCG_CAP [24] is 0, any uncorrected error condition (VAL =1 and UC=1) including the one with the EN flag cleared are fatal and the handler must signal the operating system to reset the system. For the errors that do not generate machine check exceptions, the EN flag has no meaning.
- When the VAL flag is one, the UC flag is one, the EN flag is one and the PCC flag is zero in the IA32_MCi_STATUS register, the error in this bank is an uncorrected recoverable (UCR) error. The MCE handler needs to examine the S flag and the AR flag to find the type of the UCR error for software recovery and determine if software error recovery is possible.
- When both the S and the AR flags are clear in the IA32_MCi_STATUS register for the UCR error (VAL=1, UC=1, EN=x and PCC=0), the error in this bank is an uncorrected no-action required error (UCNA). UCNA errors are uncorrected but do not require any OS recovery action to continue execution. These errors indicate that some data in the system is corrupt, but that data has not been consumed and may not be consumed. If that data is consumed a non-UCNA machine check exception will be generated. UCNA errors are signaled in the same way as corrected machine check errors and the CMCI and CMC polling handler is primarily responsible for handling UCNA errors. Like corrected errors, the MCA handler can optionally log and clear UCNA errors as long as it can avoid the undesired race condition with the CMCI or CMC polling handler. As UCNA errors are not the source of machine check exceptions, the MCA handler should continue searching for uncorrected or software recoverable errors in all other MC banks.
- When the S flag in the IA32_MCi_STATUS register is set for the UCR error ((VAL=1, UC=1, EN=1 and PCC=0), the error in this bank is software recoverable and it was signaled through a machine-check exception. The AR flag in the IA32_MCi_STATUS register further clarifies the type of the software recoverable errors.
- When the AR flag in the IA32_MCi_STATUS register is clear for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action optional (SRAO) error. The MCE handler and the operating system can analyze the IA32_MCi_STATUS [15:0] to implement MCA error code specific optional recovery action, but this recovery action is optional. System software can resume the program execution from the instruction pointer saved on the stack for the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.
- Even if the OVER flag in the IA32_MCi_STATUS register is set for the SRAO error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=0), the MCE handler can take recovery action for the SRAO error logged in the IA32_MCi_STATUS register. Since the recovery action for SRAO errors is optional, restarting the program execution from the instruction pointer saved on the stack for the machine check exception is still possible for the overflowed SRAO error if the RIPV flag in the IA32_MCG_STATUS is set.
- When the AR flag in the IA32_MCi_STATUS register is set for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action required (SRAR) error. The MCE handler and the operating system must take recovery action in order to continue execution after the machine-check exception. The MCA handler and the operating system need to analyze the IA32_MCi_STATUS [15:0] to determine the MCA error code specific recovery action. If no recovery action can be performed, the operating system must reset the system.
- When the OVER flag in the IA32_MCi_STATUS register is set for the SRAR error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=1), the MCE handler cannot take recovery action as the information of the SRAR error in the IA32_MCi_STATUS register was potentially lost due to the overflow condition. Since the recovery action for SRAR errors must be taken, the MCE handler must signal the operating system to reset the system.
- When the MCE handler cannot find any uncorrected (VAL=1, UC=1 and EN=1) or any software recoverable errors (VAL=1, UC=1, EN=1, PCC=0 and S=1) in any of the IA32_MCi banks of the processors, this is an unexpected condition for the MCE handler and the handler should signal the operating system to reset the system.
- Before returning from the machine-check exception handler, software must clear the MCIP flag in the IA32_MCG_STATUS register. The MCIP flag is used to detect recursion. The machine-check architecture does not support recursion. When the processor receives a machine check when MCIP is set, it automatically enters the shutdown state.

Example 15-4 gives pseudocode for an MC exception handler that supports recovery of UCR.

Example 15-4. Machine-Check Error Handler Pseudocode Supporting UCR

```

MACHINE CHECK HANDLER: (* Called from INT 18 handler *)
NOERROR = TRUE;
ProcessorCount = 0;
IF CPU supports MCA
  THEN
    RESTARTABILITY = TRUE;
    IF (Processor Family = 6 AND DisplayModel ≥ 0EH) OR (Processor Family > 6)
      THEN
        IF ( MCG_LMCE = 1)
          MCA_BROADCAST = FALSE;
        ELSE
          MCA_BROADCAST = TRUE;
        FI;
        Acquire SpinLock;
        ProcessorCount++; (* Allowing one logical processor at a time to examine machine check registers *)
        CALL MCA ERROR PROCESSING; (* returns RESTARTABILITY and NOERROR *)
      ELSE
        MCA_BROADCAST = FALSE;
        (* Implement a rendezvous mechanism with the other processors if necessary *)
        CALL MCA ERROR PROCESSING;
    FI;
  ELSE (* Pentium(R) processor compatible *)
    READ P5_MC_ADDR
    READ P5_MC_TYPE;
    RESTARTABILITY = FALSE;
FI;

IF NOERROR = TRUE
  THEN
    IF NOT (MCG_RIPV = 1 AND MCG_EIPV = 0)
      THEN
        RESTARTABILITY = FALSE;
    FI
  FI;

IF RESTARTABILITY = FALSE
  THEN
    Report RESTARTABILITY to console;
    Reset system;
  FI;

IF MCA_BROADCAST = TRUE
  THEN
    IF ProcessorCount = MAX_PROCESSORS
      AND NOERROR = TRUE
      THEN
        Report RESTARTABILITY to console;
        Reset system;
    FI;
    Release SpinLock;
    Wait till ProcessorCount = MAX_PROCESSORS on system;
    (* implement a timeout and abort function if necessary *)
  FI;
CLEAR IA32_MCG_STATUS;
RESUME Execution;
(* End of MACHINE CHECK HANDLER*)

```

```

MCA ERROR PROCESSING: (* MCA Error Processing Routine called from MCA Handler *)
IF MCIP flag in IA32_MCG_STATUS = 0
  THEN (* MCIP=0 upon MCA is unexpected *)
    RESTARTABILITY = FALSE;
  FI;

```


MACHINE-CHECK ARCHITECTURE

FOR each bank of machine-check registers

DO

CLEAR_MC_BANK = FALSE;

READ IA32_MCI_STATUS;

IF VAL Flag in IA32_MCI_STATUS = 1

THEN

IF UC Flag in IA32_MCI_STATUS = 1

THEN

IF Bit 24 in IA32_MCG_CAP = 0

THEN (* the processor does not support software error recovery *)

RESTARTABILITY = FALSE;

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI;

(* the processor supports software error recovery *)

IF EN Flag in IA32_MCI_STATUS = 0 AND OVER Flag in IA32_MCI_STATUS=0

THEN (* It is a spurious MCA Log. Log and clear the register *)

CLEAR_MC_BANK = TRUE;

GOTO LOG MCA REGISTER;

FI;

IF PCC = 1 and EN = 1 in IA32_MCI_STATUS

THEN (* processor context might have been corrupted *)

RESTARTABILITY = FALSE;

ELSE (* It is a uncorrected recoverable (UCR) error *)

IF S Flag in IA32_MCI_STATUS = 0

THEN

IF AR Flag in IA32_MCI_STATUS = 0

THEN (* It is a uncorrected no action required (UCNA) error *)

GOTO CONTINUE; (* let CMCI and CMC polling handler to process *)

ELSE

RESTARTABILITY = FALSE; (* S=0, AR=1 is illegal *)

FI

FI;

IF RESTARTABILITY = FALSE

THEN (* no need to take recovery action if RESTARTABILITY is already false *)

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI;

(* S in IA32_MCI_STATUS = 1 *)

IF AR Flag in IA32_MCI_STATUS = 1

THEN (* It is a software recoverable and action required (SRAR) error *)

IF OVER Flag in IA32_MCI_STATUS = 1

THEN

RESTARTABILITY = FALSE;

NOERROR = FALSE;

GOTO LOG MCA REGISTER;

FI

IF MCACOD Value in IA32_MCI_STATUS is recognized

AND Current Processor is an Affected Processor

THEN

Implement MCACOD specific recovery action;

CLEAR_MC_BANK = TRUE;

ELSE

RESTARTABILITY = FALSE;

FI;

ELSE (* It is a software recoverable and action optional (SRAO) error *)

IF OVER Flag in IA32_MCI_STATUS = 0 AND

MCACOD in IA32_MCI_STATUS is recognized

THEN

Implement MCACOD specific recovery action;

FI;

CLEAR_MC_BANK = TRUE;

FI; AR

FI; PCC

NOERROR = FALSE;

```

        GOTO LOG MCA REGISTER;
    ELSE (* It is a corrected error; continue to the next IA32_MCi_STATUS *)
        GOTO CONTINUE;
    FI; UC
    FI; VAL
LOG MCA REGISTER:
    SAVE IA32_MCi_STATUS;
    If MISCV in IA32_MCi_STATUS
        THEN
            SAVE IA32_MCi_MISC;
        FI;
    IF ADDRv in IA32_MCi_STATUS
        THEN
            SAVE IA32_MCi_ADDR;
        FI;
    IF CLEAR_MC_BANK = TRUE
        THEN
            SET all 0 to IA32_MCi_STATUS;
            If MISCV in IA32_MCi_STATUS
                THEN
                    SET all 0 to IA32_MCi_MISC;
                FI;
            IF ADDRv in IA32_MCi_STATUS
                THEN
                    SET all 0 to IA32_MCi_ADDR;
                FI;
        FI;
    CONTINUE:
    OD;
(*END FOR *)
RETURN;
(* End of MCA ERROR PROCESSING*)

```

15.10.4.2 Corrected Machine-Check Handler for Error Recovery

When writing a corrected machine check handler, which is invoked as a result of CMCI or called from an OS CMC Polling dispatcher, consider the following:

- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank does not contain valid error information and does not need to be checked.
- The CMCI or CMC polling handler is responsible for logging and clearing corrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or not (UC=1).
- When IA32_MCG_CAP [24] is one, the CMC handler is also responsible for logging and clearing uncorrected no-action required (UCNA) errors. When the UC flag is one but the PCC, S, and AR flags are zero in the IA32_MCi_STATUS register, the reported error in this bank is an uncorrected no-action required (UCNA) error. In cases when SRAO error are signaled as UCNA error via CMCI, software can perform recovery for those errors identified in Table 15-16.
- In addition to corrected errors and UCNA errors, the CMC handler optionally logs uncorrected (UC=1 and PCC=1), software recoverable machine check errors (UC=1, PCC=0 and S=1), but should avoid clearing those errors from the MC banks. Clearing these errors may result in accidentally removing these errors before these errors are actually handled and processed by the MCE handler for attempted software error recovery.

Example 15-5 gives pseudocode for a CMCI handler with UCR support.

Example 15-5. Corrected Error Handler Pseudocode with UCR Support

Corrected Error HANDLER: (* Called from CMCI handler or OS CMC Polling Dispatcher*)
 IF CPU supports MCA

```

  THEN
    FOR each bank of machine-check registers
      DO
        READ IA32_MCI_STATUS;
        IF VAL flag in IA32_MCI_STATUS = 1
          THEN
            IF UC Flag in IA32_MCI_STATUS = 0 (* It is a corrected error *)
              THEN
                GOTO LOG CMC ERROR;
            ELSE
              IF Bit 24 in IA32_MCG_CAP = 0
                THEN
                  GOTO CONTINUE;
              FI;
              IF S Flag in IA32_MCI_STATUS = 0 AND AR Flag in IA32_MCI_STATUS = 0
                THEN (* It is a uncorrected no action required error *)
                  GOTO LOG CMC ERROR
              FI
              IF EN Flag in IA32_MCI_STATUS = 0
                THEN (* It is a spurious MCA error *)
                  GOTO LOG CMC ERROR
              FI;
            FI;
          FI;
          GOTO CONTINUE;
        LOG CMC ERROR:
        SAVE IA32_MCI_STATUS;
        If MISCV Flag in IA32_MCI_STATUS
          THEN
            SAVE IA32_MCI_MISC;
            SET all 0 to IA32_MCI_MISC;
          FI;
        IF ADDR_V Flag in IA32_MCI_STATUS
          THEN
            SAVE IA32_MCI_ADDR;
            SET all 0 to IA32_MCI_ADDR
          FI;
        SET all 0 to IA32_MCI_STATUS;
        CONTINUE:
      OD;
    (*END FOR *)
  FI;

```

23. Updates to Chapter 16, Volume 3B

Change bars and green text show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

Changes to this chapter: Added additional information in Section 16.11.3 "Integrated Memory Controller Machine Check Errors". Typo corrections as needed.

CHAPTER 16

INTERPRETING MACHINE-CHECK ERROR CODES

Encoding of the model-specific and other information fields is different across processor families. The differences are documented in the following sections.

16.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H MACHINE ERROR CODES FOR MACHINE CHECK

Section 16.1 provides information for interpreting additional model-specific fields for external bus errors relating to processor family 06H. The references to processor family 06H refers to only IA-32 processors with CPUID signatures listed in Table 16-1.

Table 16-1. CPUID DisplayFamily_DisplayModel Signatures for Processor Family 06H

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_09H	Intel Pentium M processor
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

These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes. Incremental decoding information is listed in Table 16-2.

Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0		
Model specific errors	18:16	Reserved	Reserved
Model specific errors	24:19	Bus queue request type	000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error

Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
			001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSH2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error
Model specific errors	27:25	Bus queue error type	000 for BQ_ERR_HARD_TYPE error 001 for BQ_ERR_DOUBLE_TYPE error 010 for BQ_ERR_AERR2_TYPE error 100 for BQ_ERR_SINGLE_TYPE error 101 for BQ_ERR_AERR1_TYPE error
Model specific errors	28	FRC error	1 if FRC error active
	29	BERR	1 if BERR is driven
	30	Internal BINIT	1 if BINIT driven for this processor
	31	Reserved	Reserved
Other information	34:32	Reserved	Reserved
	35	External BINIT	1 if BINIT is received from external bus.
	36	Response parity error	This bit is asserted in IA32_MC _i _STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	Bus BINIT	This bit is asserted in IA32_MC _i _STATUS if this component has received a hard error response on a split transaction one access that has needed to be split across the 64-bit external bus interface into two accesses).
	38	Timeout BINIT	This bit is asserted in IA32_MC _i _STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. ² The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard error	This bit is asserted in IA32_MC _i _STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.

Table 16-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
	43	IERR	This bit is asserted in IA32_MCi_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.
	44	AERR	This bit is asserted in IA32_MCi_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors AERR asserted). While this bit is asserted, it cannot be overwritten.
	45	UECC	The Uncorrectable ECC error bit is asserted in IA32_MCi_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.
	46	CECC	The correctable ECC error bit is asserted in IA32_MCi_STATUS for corrected ECC errors.
	54:47	ECC syndrome	The ECC syndrome field in IA32_MCi_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MCi_STATUS. A previous valid ECC error in IA32_MCi_STATUS is indicated by IA32_MCi_STATUS.bit45 (uncorrectable error occurred) being asserted. After processing an ECC error, machine-check handling software should clear IA32_MCi_STATUS.bit45 so that future ECC error syndromes can be logged.
	56:55	Reserved	Reserved.
Status register validity indicators ¹	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. For processors with a CPUID signature of 06_OEH, a ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit.

16.2 INCREMENTAL DECODING INFORMATION: INTEL CORE 2 PROCESSOR FAMILY MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-4 provides information for interpreting additional model-specific fields for external bus errors relating to processor based on Intel Core microarchitecture, which implements the P4 bus specification. Table 16-3 lists the CPUID signatures for Intel 64 processors that are covered by Table 16-4. These errors are reported in the IA32_MCi_STATUS MSR. They are reported architecturally as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes.

Table 16-3. CPUID DisplayFamily_DisplayModel Signatures for Processors Based on Intel Core Microarchitecture

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_1DH	Intel Xeon Processor 7400 series.
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9650.
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors.

Table 16-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel Core Microarchitecture

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0		
Model specific errors	18:16	Reserved	Reserved
Model specific errors	24:19	Bus queue request type	'000001 for BQ_PREF_READ_TYPE error 000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error 100100 for BQ_L2_WI_RFO_TYPE error 100110 for BQ_L2_WI_ITOM_TYPE error
Model specific errors	27:25	Bus queue error type	'001 for Address Parity Error '010 for Response Hard Error '011 for Response Parity Error
Model specific errors	28	MCE Driven	1 if MCE is driven
	29	MCE Observed	1 if MCE is observed
	30	Internal BINIT	1 if BINIT driven for this processor
	31	BINIT Observed	1 if BINIT is observed for this processor
Other information	33:32	Reserved	Reserved
	34	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	35	Reserved	Reserved

**Table 16-4. Incremental Bus Error Codes of Machine Check for Processors
Based on Intel Core Microarchitecture (Contd.)**

Type	Bit No.	Bit Function	Bit Description
	36	Response parity error	This bit is asserted in IA32_MC _i _STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	38	Timeout BINIT	This bit is asserted in IA32_MC _i _STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard error	This bit is asserted in IA32_MC _i _STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC _i _STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.
	44	Reserved	Reserved
	45	Reserved	Reserved
	46	Reserved	Reserved
	54:47	Reserved	Reserved
	56:55	Reserved	Reserved.
Status register validity indicators ¹	63:57		

NOTES:

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

16.2.1 Model-Specific Machine Check Error Codes for Intel Xeon Processor 7400 Series

Intel Xeon processor 7400 series has machine check register banks that generally follows the description of Chapter 15 and Section 16.2. Additional error codes specific to Intel Xeon processor 7400 series is describe in this section.

MC4_STATUS[63:0] is the main error logging for the processor's L3 and front side bus errors for Intel Xeon processor 7400 series. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

16.2.1.1 Processor Machine Check Status Register Incremental MCA Error Code Definition

Intel Xeon processor 7400 series use compound MCA Error Codes for logging its Bus internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines incremental Machine Check error types (IA32_MC6_STATUS[15:0]) beyond those defined in Chapter 15. Table 16-5 lists these incremental MCA error code types that apply to IA32_MC6_STATUS. Error code details are specified in MC6_STATUS [31:16] (see Section 16.2.2), the "Model Specific Error Code" field. The information in the "Other_Info" field (MC4_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC6_MISC register format.

Table 16-5. Incremental MCA Error Code Types for Intel Xeon Processor 7400

Processor MCA_Error_Code (MC6_STATUS[15:0])			
Type	Error Code	Binary Encoding	Meaning
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 101x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 110x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 1110 0000 1111	Bus and Interconnection Error Type Code
		0000 1111 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations

The **Bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

16.2.2 Intel Xeon Processor 7400 Model Specific Error Code Field

16.2.2.1 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC6_STATUS (bits 31:16).

Table 16-6. Type B Bus and Interconnect Error Codes

Bit Num	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase
19:17		Reserved
20	FSB Hard Fail Response	"Hard Failure" response received for a local transaction
21	FSB Response Parity	Parity error on FSB response field detected
22	FSB Data Parity	FSB data parity error on inbound data detected
31:23	---	Reserved

16.2.2.2 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

Table 16-7. Type C Cache Bus Controller Error Codes

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0
0000_0000_0000_0010 0002H	Inclusion Error from Core 1
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1
0000_0000_0000_0101 0005H	Inclusion Error from FSB
0000_0000_0000_0110 0006H	SNP Stall Error from FSB
0000_0000_0000_0111 0007H	Write Stall Error from FSB
0000_0000_0000_1000 0008H	FSB Arb Timeout Error
0000_0000_0000_1010 000AH	Inclusion Error from Core 2
0000_0000_0000_1011 000BH	Write Exclusive Error from Core 2
0000_0010_0000_0000 0200H	Internal Timeout error
0000_0011_0000_0000 0300H	Internal Timeout Error
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow
0000_0101_0000_0000 0500H	Quiet cycle Timeout Error (correctable)
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data
1100_0000_0000_1000 C008H	Correctable ECC event on outgoing Core 2 data
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data
1110_0000_0000_1000 E008H	Uncorrectable ECC error on outgoing Core 2 data
— all other encodings —	Reserved

16.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-8 through Table 16-12 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor 3400, 3500, 5500 series with CPUID DisplayFamily_DisplaySignature 06_1AH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC0 and IA32_MC1, incremental error codes for internal machine check is reported in the register bank IA32_MC7, and incremental error codes for the memory controller unit is reported in the register banks IA32_MC8.

16.3.1 Intel QPI Machine Check Errors

Table 16-8. Intel QPI Machine Check Error Codes for IA32_MC0_STATUS and IA32_MC1_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	16	Header Parity	If 1, QPI Header had bad parity
	17	Data Parity	If 1, QPI Data packet had bad parity
	18	Retries Exceeded	If 1, number of QPI retries was exceeded
	19	Received Poison	If 1, Received a data packet that was marked as poisoned by the sender
	21:20	Reserved	Reserved
	22	Unsupported Message	If 1, QPI received a message encoding it does not support
	23	Unsupported Credit	If 1, QPI credit type is not supported.
	24	Receive Flit Overrun	If 1, Sender sent too many QPI flits to the receiver.
	25	Received Failed Response	If 1, Indicates that sender sent a failed response to receiver.
	26	Receiver Clock Jitter	If 1, clock jitter detected in the internal QPI clocking
	56:27	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-9. Intel QPI Machine Check Error Codes for IA32_MC0_MISC and IA32_MC1_MISC

Type	Bit No.	Bit Function	Bit Description
Model specific errors ¹	7:0	QPI Opcode	Message class and opcode from the packet with the error
	13:8	RTID	QPI Request Transaction ID
	15:14	Reserved	Reserved
	18:16	RHNID	QPI Requestor/Home Node ID
	23:19	Reserved	Reserved
	24	IIB	QPI Interleave/Head Indication Bit

NOTES:

1. Which of these fields are valid depends on the error type.

16.3.2 Internal Machine Check Errors

Table 16-10. Machine Check Error Codes for IA32_MC7_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
Model specific errors			

Type	Bit No.	Bit Function	Bit Description
	23:16	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 03h - Reset firmware did not complete 08h - Received an invalid CMPD 0Ah - Invalid Power Management Request 0Dh - Invalid S-state transition 11h - VID controller does not match POC controller selected 1Ah - MSID from POC does not match CPU MSID
	56:32	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.3.3 Memory Controller Errors

Table 16-11. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory error format: 1MMMCCCC
Model specific errors			
	16	Read ECC error	If 1, ECC occurred on a read
	17	RAS ECC error	If 1, ECC occurred on a scrub
	18	Write parity error	If 1, bad parity on a write
	19	Redundancy loss	If 1, Error in half of redundant memory
	20	Reserved	Reserved
	21	Memory range error	If 1, Memory access out of range
	22	RTID out of range	If 1, Internal ID invalid
	23	Address parity error	If 1, bad address parity
	24	Byte enable parity error	If 1, bad enable parity
Other information	37:25	Reserved	Reserved
	52:38	CORE_ERR_CNT	Corrected error count
	56:53	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-12. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_MISC

Type	Bit No.	Bit Function	Bit Description
Model specific errors ¹			
	7:0	RTId	Transaction Tracker ID
	15:8	Reserved	Reserved
	17:16	DIMM	DIMM ID which got the error
	19:18	Channel	Channel ID which got the error
	31:20	Reserved	Reserved
	63:32	Syndrome	ECC Syndrome

NOTES:

1. Which of these fields are valid depends on the error type.

16.4 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-13 through Table 16-15 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor E5 Family with CPUID DisplayFamily_DisplaySignature 06_2DH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC6 and IA32_MC7, incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, and incremental error codes for the memory controller unit is reported in the register banks IA32_MC8-IA32_MC11.

16.4.1 Internal Machine Check Errors

Table 16-13. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - L_Parity_Error 0011b - Bad_OpCode 0100b - L_Stack_Underflow 0101b - L_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MC_IMC_FORCE_SR_S3_TIMEOUT 0Eh - MC_CPD_UNCPD_ST_TIMEOUT 0Fh - MC_PKGS_SAFE_WP_TIMEOUT 43h - MC_PECI_MAILBOX QUIESCE_TIMEOUT 5Ch - MC_MORE_THAN_ONE_LT_AGENT 60h - MC_INVALID_PKGS_REQ_PCH 61h - MC_INVALID_PKGS_REQ_QPI 62h - MC_INVALID_PKGS_RES_QPI 63h - MC_INVALID_PKGC_RES_PCH 64h - MC_INVALID_PKG_STATE_CONFIG 70h - MC_WATCHDG_TIMEOUT_PKGC_SLAVE 71h - MC_WATCHDG_TIMEOUT_PKGC_MASTER 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 7ah - MC_HA_FAILSTS_CHANGE_DETECTED 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.4.2 Intel QPI Machine Check Errors

Table 16-14. Intel QPI MC Error Codes for IA32_MC6_STATUS and IA32_MC7_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors			
	56:16	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.4.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC8_STATUS-IA32_MC11_STATUS. The supported error codes are follows the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture,"). MSR_ERROR_CONTROL.[bit 1] can enable additional informa-

tion logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC; (i = 8, 11).

Table 16-15. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 8, 11)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	31:16	Reserved except for the following	001H - Address parity error 002H - HA Wrt buffer Data parity error 004H - HA Wrt byte enable parity error 008H - Corrected patrol scrub error 010H - Uncorrected patrol scrub error 020H - Corrected spare error 040H - Uncorrected spare error
Model specific errors	36:32	Other info	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first device error when corrected error is detected during normal read.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-16. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 8, 11)

Type	Bit No.	Bit Function	Bit Description
MCA addr info ¹	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		<ul style="list-style-type: none"> When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second device error when corrected error is detected during normal read. Otherwise contain parity error if MCI_Status indicates HA_WB_Data or HA_W_BE parity error.
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error failing rank.
	58:56	Reserved	Reserved
	61:59	Reserved	Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data from the first correctable error in a memory device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

NOTES:

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

16.5 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v2 family and the Intel® Xeon® processor E7 v2 family are based on the Ivy Bridge-EP microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3EH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, Table lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5. Information listed in Table 16-14 for QPI MC error code apply to IA32_MC5_STATUS. Incremental error codes for the memory controller unit is reported in the register banks IA32_MC9-IA32_MC16. Table 16-18 lists model-specific error codes apply to IA32_MCi_STATUS, i = 9-16.

16.5.1 Internal Machine Check Errors

Table 16-17. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - I_Parity_Error 0011b - Bad_OpCode 0100b - I_Stack_Underflow 0101b - I_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MC_IMC_FORCE_SR_S3_TIMEOUT 0Eh - MC_CPD_UNCPD_ST_TIMEOUT 0Fh - MC_PKGS_SAFE_WP_TIMEOUT 43h - MC_PECI_MAILBOX QUIESCE_TIMEOUT 44h - MC_CRITICAL_VR_FAILED 45h - MC_ICC_MAX-NOTSUPPORTED 5Ch - MC_MORE_THAN_ONE_LT_AGENT 60h - MC_INVALID_PKGS_REQ_PCH 61h - MC_INVALID_PKGS_REQ_QPI 62h - MC_INVALID_PKGS_RES_QPI 63h - MC_INVALID_PKGC_RES_PCH 64h - MC_INVALID_PKG_STATE_CONFIG 70h - MC_WATCHDG_TIMEOUT_PKGC_SLAVE 71h - MC_WATCHDG_TIMEOUT_PKGC_MASTER 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER

Type	Bit No.	Bit Function	Bit Description
			7Ah - MC_HA_FAILSTS_CHANGE_DETECTED 7Bh - MC_PCIE_R2PCIE-RW_BLOCK_ACK_TIMEOUT 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.5.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC9_STATUS-IA32_MC16_STATUS. The supported error codes are follows the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC; (i = 9-16).

IA32_MCi_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32_MCi_STATUS (i=13-16).

Table 16-18. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9-16)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 000F 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	001H - Address parity error 002H - HA Wrt buffer Data parity error 004H - HA Wrt byte enable parity error 008H - Corrected patrol scrub error
			010H - Uncorrected patrol scrub error 020H - Corrected spare error 040H - Uncorrected spare error 080H - Corrected memory read error. (Only applicable with iMC’s “Additional Error logging” Mode-1 enabled.) 100H - iMC, WDB, parity errors
	36:32	Other info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 15, “Machine-Check Architecture,”
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-19. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9-16)

Type	Bit No.	Bit Function	Bit Description
MCA addr info ¹	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		If the error logged is MCWrDataPar error or MCWrBEPAr error, this field is the WDB ID that has the parity error. OR if the second error logged is a correctable read error, MC logs the second error device in this field.
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56		Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

NOTES:

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

16.5.3 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32_MC8_{STATUS,ADDR,MISC}.

16.6 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v3 family is based on the Haswell-E microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3FH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, Table 16-20 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Information listed in Table 16-21 for QPI MC error codes. Incremental error codes for the memory controller unit is reported in the register banks IA32_MC9-IA32_MC16. Table 16-22 lists model-specific error codes apply to IA32_MCi_STATUS, i = 9-16.

16.6.1 Internal Machine Check Errors

Table 16-20. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
MCACOD ²	15:0	Internal Errors	0402h - PCU internal Errors 0403h - PCU internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors. On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00xxb - PCU internal error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 09h - MC_MESSAGE_CHANNEL_TIMEOUT 13h - MC_DMI_TRAINING_TIMEOUT 15h - MC_DMI_CPU_RESET_ACK_TIMEOUT 1Eh - MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25h - MC_SVID_COMMAND_TIMEOUT 29h - MC_VR_VOUT_MAC_LT_FUSED_SVID 2Bh - MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2Ch - MC_PKGC_WATCHDOG_HANG_CBZ_UP 44h - MC_CRITICAL_VR_FAILED 46h - MC_VID_RAMP_DOWN_FAILED 49h - MC_SVID_WRITE_REG_VOUT_MAX_FAILED 4Bh - MC_BOOT_VID_TIMEOUT. Timeout setting boot VID for DRAM 0. 4Fh - MC_SVID_COMMAND_ERROR. 52h - MC_FIVR_CATAS_OVERVOL_FAULT. 53h - MC_FIVR_CATAS_OVERCUR_FAULT. 57h - MC_SVID_PKGC_REQUEST_FAILED 58h - MC_SVID_IMON_REQUEST_FAILED 59h - MC_SVID_ALERT_REQUEST_FAILED 62h - MC_INVALID_PKGS_RSP_QPI 64h - MC_INVALID_PKG_STATE_CONFIG 67h - MC_HA_IMC_Rw_BLOCK_ACK_TIMEOUT 6Ah - MC_MSGCH_PMREQ_CMP_TIMEOUT 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

2. The internal error codes may be model-specific.

16.6.2 Intel QPI Machine Check Errors

MC error codes associated with the Intel QPI agents are reported in the MSRs IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, “Machine-Check Architecture,”).

Table 16-21 lists model-specific fields to interpret error codes applicable to IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS.

Table 16-21. Intel QPI MC Error Codes for IA32_MCi_STATUS (i = 5, 20, 21)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors	31:16	MSCOD	02h - Intel QPI physical layer detected drift buffer alarm.
			03h - Intel QPI physical layer detected latency buffer rollover.
			10h - Intel QPI link layer detected control error from R3QPI.
			11h - Rx entered LLR abort state on CRC error.
			12h - Unsupported or undefined packet.
			13h - Intel QPI link layer control error.
			15h - RBT used un-initialized value.
			20h - Intel QPI physical layer detected a QPI in-band reset but aborted initialization
			21h - Link failover data self-healing
			22h - Phy detected in-band reset (no width change).
			23h - Link failover clock failover
			30h -Rx detected CRC error - successful LLR after Phy re-init.
			31h -Rx detected CRC error - successful LLR without Phy re-init.
			All other values are reserved.
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.6.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC9_STATUS-IA32_MC16_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture,”).

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC; (i = 9-16).

IA32_MCi_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32_MCi_STATUS (i=13-16).

Table 16-22. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9-16)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error 0002H - Uncorrected HA write data error 0004H - Uncorrected HA data byte enable error 0008H - Corrected patrol scrub error 0010H - Uncorrected patrol scrub error 0020H - Corrected spare error 0040H - Uncorrected spare error 0080H - Corrected memory read error. (Only applicable with iMC's "Additional Error logging" Mode-1 enabled.) 0100H - iMC, write data buffer parity errors 0200H - DDR4 command address parity error
	36:32	Other info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-23. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9-16)

Type	Bit No.	Bit Function	Bit Description
MCA addr info ¹	8:0		See Chapter 15, "Machine-Check Architecture,"
Model specific errors	13:9		If the error logged is MCWrDataPar error or MCWrBEPPar error, this field is the WDB ID that has the parity error. OR if the second error logged is a correctable read error, MC logs the second error device in this field.
Model specific errors	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
Model specific errors	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56		Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

NOTES:

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

16.6.4 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32_MC8_{STATUS,ADDR,MISC}.

16.7 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor D family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_56H. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, Table 16-24 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental error codes for the memory controller unit is reported in the register banks IA32_MC9-IA32_MC10. Table 16-18 lists model-specific error codes apply to IA32_MCi_STATUS, i = 9-10.

16.7.1 Internal Machine Check Errors

Table 16-24. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
MCACOD ²	15:0	internal Errors	0402h - PCU internal Errors 0403h - internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors. On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00x1b - PCU internal error 001xb - PCU internal error

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved except for the following	x1xxb - UBOX error
	31:24	Reserved except for the following	00h - No Error 09h - MC_MESSAGE_CHANNEL_TIMEOUT 13h - MC_DMI_TRAINING_TIMEOUT 15h - MC_DMI_CPU_RESET_ACK_TIMEOUT 1Eh - MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25h - MC_SVID_COMMAND_TIMEOUT 26h - MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 29h - MC_VR_VOUT_MAC_LT_FUSED_SVID 2Bh - MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2Ch - MC_PKGC_WATCHDOG_HANG_CBZ_UP 44h - MC_CRITICAL_VR_FAILED 46h - MC_VID_RAMP_DOWN_FAILED 49h - MC_SVID_WRITE_REG_VOUT_MAX_FAILED 4Bh - MC_PP1_BOOT_VID_TIMEOUT. Timeout setting boot VID for DRAM 0. 4Fh - MC_SVID_COMMAND_ERROR. 52h - MC_FIVR_CATAS_OVERVOL_FAULT. 53h - MC_FIVR_CATAS_OVERCUR_FAULT. 57h - MC_SVID_PKGC_REQUEST_FAILED 58h - MC_SVID_IMON_REQUEST_FAILED 59h - MC_SVID_ALERT_REQUEST_FAILED 62h - MC_INVALID_PKGS_RSP_QPI 64h - MC_INVALID_PKG_STATE_CONFIG 67h - MC_HA_IMC_Rw_BLOCK_ACK_TIMEOUT 6Ah - MC_MSGCH_PMREQ_CMP_TIMEOUT 72h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.

16.7.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC9_STATUS-IA32_MC10_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture,").

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC; (i = 9-10).

Table 16-25. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9-10)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error
			0002H - Uncorrected HA write data error
			0004H - Uncorrected HA data byte enable error
			0008H - Corrected patrol scrub error
			0010H - Uncorrected patrol scrub error
			0100H - iMC, write data buffer parity errors
			0200H - DDR4 command address parity error
	36:32	Other info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators ¹	63:57		

NOTES:

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

16.8 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v4 family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_4FH. Incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, Table 16-20 in Section 16.6.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Information listed in Table 16-21 of Section 16.6.1 covers QPI MC error codes.

16.8.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC9_STATUS-IA32_MC16_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

Table 16-26 lists model-specific error codes apply to IA32_MCi_STATUS, i = 9-16.

IA32_MCi_STATUS (i=9-12) log errors from the first memory controller. The second memory controller logs into IA32_MCi_STATUS (i=13-16).

Table 16-26. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9-16)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - DDR3 address parity error 0002H - Uncorrected HA write data error 0004H - Uncorrected HA data byte enable error 0008H - Corrected patrol scrub error 0010H - Uncorrected patrol scrub error 0020H - Corrected spare error 0040H - Uncorrected spare error 0100H - iMC, write data buffer parity errors 0200H - DDR4 command address parity error
	36:32	Other info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators ¹	63:57		

NOTES:

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

16.8.2 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32_MC7_MISC and IA32_MC8_MISC. Table 16-27 lists model-specific error codes apply to IA32_MCi_MISC, i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32_MC8_{STATUS,ADDR,MISC}.

Table 16-27. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
40:9	Reserved	Reserved
41	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.
42	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63:43	Reserved	Reserved

16.9 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR SCALABLE FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the Intel® Xeon® Processor Scalable Family with CPUID DisplayFamily_DisplaySignature 06_55H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 16-28 in Section 16.9.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

16.9.1 Internal Machine Check Errors

Table 16-28. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
MCACOD ²	15:0	Internal Errors	0402h - PCU internal Errors 0403h - PCU internal Errors 0406h - Intel TXT Errors 0407h - Other UBOX internal Errors. On an IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS (After a 3-strike, the core MCA banks will be unavailable).
Model specific errors	19:16	Reserved except for the following	0000b - No Error 00xxb - PCU internal error
	23:20	Reserved	Reserved
	31:24	Reserved except for the following	00h - No Error 0Dh - MCA_DMI_TRAINING_TIMEOUT 0Fh - MCA_DMI_CPU_RESET_ACK_TIMEOUT 10h - MCA_MORE_THAN_ONE_LT_AGENT 1Eh - MCA_BIOS_RST_CPL_INVALID_SEQ 1Fh - MCA_BIOS_INVALID_PKG_STATE_CONFIG 25h - MCA_MESSAGE_CHANNEL_TIMEOUT 27h - MCA_MSGCH_PMREQ_CMP_TIMEOUT 30h - MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 31h - MCA_PKGC_INVALID_RSP_PCH 33h - MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 34h - MCA_PKGC_WATCHDOG_HANG_CBZ_UP 38h - MCA_PKGC_WATCHDOG_HANG_C3_UP_SF 40h - MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 41h - MCA_SVID_COMMAND_TIMEOUT 42h - MCA_SVID_VCCIN_VR_VOUT_MAX_FAILURE 43h - MCA_SVID_CPU_VR_CAPABILITY_ERROR 44h - MCA_SVID_CRITICAL_VR_FAILED 45h - MCA_SVID_SA_ITD_ERROR 46h - MCA_SVID_READ_REG_FAILED 47h - MCA_SVID_WRITE_REG_FAILED 48h - MCA_SVID_PKGC_INIT_FAILED

Type	Bit No.	Bit Function	Bit Description
			49h - MCA_SVID_PKG_CONFIG_FAILED 4Ah - MCA_SVID_PKG_REQUEST_FAILED 4Bh - MCA_SVID_IMON_REQUEST_FAILED 4Ch - MCA_SVID_ALERT_REQUEST_FAILED 4Dh - MCA_SVID_MCP_VP_ABSENT_OR_RAMP_ERROR 4Eh - MCA_SVID_UNEXPECTED_MCP_VP_DETECTED 51h - MCA_FIVR_CATA_OVERVOL_FAULT 52h - MCA_FIVR_CATA_OVERCUR_FAULT 58h - MCA_WATCHDG_TIMEOUT_PKG_SLAVE 59h - MCA_WATCHDG_TIMEOUT_PKG_MASTER 5Ah - MCA_WATCHDG_TIMEOUT_PKGS_MASTER 61h - MCA_PKGS_CPD_UNPCD_TIMEOUT 63h - MCA_PKGS_INVALID_REQ_PCH 64h - MCA_PKGS_INVALID_REQ_INTERNAL 65h - MCA_PKGS_INVALID_RSP_INTERNAL 6Bh - MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	52:32	Reserved	Reserved
	54:53	CORR_ERR_STATUS	Reserved
	56:55	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.

16.9.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the MSRs IA32_MC5_STATUS, IA32_MC12_STATUS, IA32_MC19_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, "Machine-Check Architecture").

Table 16-29 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, i= 5, 12, 19.

Table 16-29. Interconnect MC Error Codes for IA32_MCi_STATUS, i = 5, 12, 19

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: - 0x0COF - Unsupported/Undefined Packet - 0x0EOF - For all other corrected and uncorrected errors

Type	Bit No.	Bit Function	Bit Description
Model specific errors	21:16	MSCOD	<p>The encoding of Uncorrectable (UC) errors are:</p> <p>00h - UC Phy Initialization Failure.</p> <p>01h - UC Phy detected drift buffer alarm.</p> <p>02h - UC Phy detected latency buffer rollover.</p> <p>10h - UC link layer Rx detected CRC error: unsuccessful LLR entered abort state</p> <p>11h - UC LL Rx unsupported or undefined packet.</p> <p>12h - UC LL or Phy control error.</p> <p>13h - UC LL Rx parameter exchange exception.</p> <p>1fh - UC LL detected control error from the link-mesh interface</p> <p>The encoding of correctable (COR) errors are:</p> <p>20h - COR Phy initialization abort</p> <p>21h - COR Phy reset</p> <p>22h - COR Phy lane failure, recovery in x8 width.</p> <p>23h - COR Phy LOc error corrected without Phy reset</p> <p>24h - COR Phy LOc error triggering Phy reset</p> <p>25h - COR Phy LOp exit error corrected with Phy reset</p> <p>30h - COR LL Rx detected CRC error - successful LLR without Phy re-init.</p> <p>31h - COR LL Rx detected CRC error - successful LLR with Phy re-init.</p> <p>All other values are reserved.</p>
	31:22	MSCOD_SPARE	<p>The definition below applies to MSCOD 12h (UC LL or Phy Control Errors)</p> <p>[Bit 22] : Phy Control Error</p> <p>[Bit 23] : Unexpected Retry.Ack flit</p> <p>[Bit 24] : Unexpected Retry.Req flit</p> <p>[Bit 25] : RF parity error</p> <p>[Bit 26] : Routeback Table error</p> <p>[Bit 27] : unexpected Tx Protocol flit (EOP, Header or Data)</p> <p>[Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow</p> <p>[Bit 29] : Link Layer Reset still in progress when Phy enters LO (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0).</p> <p>[Bit 30] : Link Layer reset initiated while protocol traffic not idle</p> <p>[Bit 31] : Link Layer Tx Parity Error</p>
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

16.9.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC13_STATUS-IA32_MC18_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

IA32_MCi_STATUS (i=13,14,17) log errors from the first memory controller. The second memory controller logs into IA32_MCi_STATUS (i=15,16,18).

Table 16-30. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13-18)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - Address parity error
			0002H - HA write data parity error
			0004H - HA write byte enable parity error
			0008H - Corrected patrol scrub error
			0010H - Uncorrected patrol scrub error
			0020H - Corrected spare error
			0040H - Uncorrected spare error
			0080H - Any HA read error
			0100H - WDB read parity error
			0200H - DDR4 command address parity error
			0400H - Uncorrected address parity error
			0800H - Unrecognized request type
			0801H - Read response to an invalid scoreboard entry
			0802H - Unexpected read response
			0803H - DDR4 completion to an invalid scoreboard entry
			0804H - Completion to an invalid scoreboard entry
			0805H - Completion FIFO overflow
			0806H - Correctable parity error
			0807H - Uncorrectable error
			0808H - Interrupt received while outstanding interrupt was not ACKed
			0809H - ERID FIFO overflow
			080aH - Error on Write credits
			080bH - Error on Read credits
			080cH - Scheduler error
			080dH - Error event
	36:32	Other info	MC logs the first error device. This is an encoded 5-bit value of the device.
	37	Reserved	Reserved
	56:38		See Chapter 15, "Machine-Check Architecture,"
Status register validity indicators ¹	63:57		

NOTES:

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

16.9.4 M2M Machine Check Errors

MC error codes associated with M2M are reported in the MSRs IA32_MC7_STATUS, IA32_MC8_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture,”).

Table 16-31. M2M MC Error Codes for IA32_MCi_STATUS (i= 7-8)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model specific errors	16	MscodDataRdErr	Logged an MC read data error
	17	Reserved	Reserved
	18	MscodPtlWrErr	Logged an MC partial write data error
	19	MscodFullWrErr	Logged a full write data error
	20	MscodBgfErr	Logged an M2M clock-domain-crossing buffer (BGF) error
	21	MscodTimeOut	Logged an M2M time out
	22	MscodParErr	Logged an M2M tracker parity error
	23	MscodBucket1Err	Logged a fatal Bucket1 error
	31:24	Reserved	Reserved
	36:32	Other info	MC logs the first error device. This is an encoded 5-bit value of the device.
37	Reserved	Reserved	
56:38		See Chapter 15, “Machine-Check Architecture,”	
Status register validity indicators ¹	63:57		

NOTES:

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

16.9.5 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32_MC7_MISC and IA32_MC8_MISC. Table 16-32 lists model-specific error codes apply to IA32_MCi_MISC, i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs to IA32_MC8_{STATUS,ADDR,MISC}.

Table 16-32. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
40:9	Reserved	Reserved
61:41	Reserved	Reserved
62	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.

16.10 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_5FH, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel® Atom® processors based on Goldmont Microarchitecture with CPUID DisplayFamily_DisplaySignature 06_5FH (code name Denverton), incremental error codes for the memory controller unit are reported in the register banks IA32_MC6 and IA32_MC7. Table 16-33 in Section 16.10.1 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, i = 6, 7.

16.10.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC6_STATUS and IA32_MC7_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

Table 16-33. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 6, 7)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	
Model specific errors	31:16	Reserved except for the following	01h - Cmd/Addr parity 02h - Corrected Demand/Patrol Scrub Error 04h - Uncorrected patrol scrub error 08h - Uncorrected demand read error 10h - WDB read ECC
	36:32	Other info	
	37	Reserved	
	56:38		See Chapter 15, “Machine-Check Architecture”.
Status register validity indicators ¹	63:57		

NOTES:

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

16.11 INCREMENTAL DECODING INFORMATION: FUTURE INTEL® XEON® PROCESSORS WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURES 06_6AH AND 06_6CH, MACHINE ERROR CODES FOR MACHINE CHECK

Future Intel® Xeon® processor with CPUID DisplayFamily_DisplaySignature of 06_6AH and 06_6CH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 16-34 in Section 16.11.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

16.11.1 Internal Machine Check Errors

Table 16-34. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
Machine Check Error Codes ¹	15:0	MCCOD	
MCCOD	15:0	Internal Errors	The value of this field will be 0402h for the PCU and 0406h for internal firmware errors. This applies for any logged error.
Model specific errors	19:16	Reserved except for the following	Model specific error code bits 19:16. This logs the type of HW UC (PCU/VCU) error that has occurred. There are 7 errors defined. 01h - Instruction address out of valid space. 02h - Double bit RAM error on Instruction Fetch. 03h - Invalid OpCode seen. 04h - Stack Underflow. 05h - Stack Overflow. 06h - Data address out of valid space. 07h - Double bit RAM error on Data Fetch.
	23:20	Reserved except for the following	Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined. 04h - Clock/power IP response timeout. 05h - SMBus controller raised SMI. 09h - PM controller received invalid transaction.
	31:24	Reserved except for the following	0Dh - MCA_LLC_BIST_ACTIVE_TIMEOUT 0Eh - MCA_DMI_TRAINING_TIMEOUT 0Fh - MCA_DMI_STRAP_SET_ARRIVAL_TIMEOUT 10h - MCA_DMI_CPU_RESET_ACK_TIMEOUT 11h - MCA_MORE_THAN_ONE_LT_AGENT 14h - MCA_INCOMPATIBLE_PCH_TYPE 1Eh - MCA_BIOS_RST_CPL_INVALID_SEQ 1Fh - MCA_BIOS_INVALID_PKG_STATE_CONFIG 2Dh - MCA_PCU_PMAX_CALIB_ERROR 2Eh - MCA_TSC100_SYNC_TIMEOUT 3Ah - MCA_GPSB_TIMEOUT 3Bh - MCA_PMSB_TIMEOUT 3Eh - MCA_IOSFSB_PMREQ_CMP_TIMEOUT 40h - MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 42h - MCA_SVID_VCCIN_VR_VOUT_FAILURE 43h - MCA_SVID_CPU_VR_CAPABILITY_ERROR 44h - MCA_SVID_CRITICAL_VR_FAILED 45h - MCA_SVID_SA_ITD_ERROR 46h - MCA_SVID_READ_REG_FAILED 47h - MCA_SVID_WRITE_REG_FAILED 4Ah - MCA_SVID_PKGC_REQUEST_FAILED

Type	Bit No.	Bit Function	Bit Description
			48h - MCA_SVID_IMON_REQUEST_FAILED 4Ch - MCA_SVID_ALERT_REQUEST_FAILED 4Dh - MCA_SVID_MCP_VR_RAMP_ERROR 56h - MCA_FIVR_PD_HARDERR 58h - MCA_WATCHDOG_TIMEOUT_PKGC_SLAVE 59h - MCA_WATCHDOG_TIMEOUT_PKGC_MASTER 5Ah - MCA_WATCHDOG_TIMEOUT_PKGS_MASTER 5Bh - MCA_WATCHDOG_TIMEOUT_MSG_CH_FSM 5Ch - MCA_WATCHDOG_TIMEOUT_BULK_CR_FSM 5Dh - MCA_WATCHDOG_TIMEOUT_IOSFSB_FSM 60h - MCA_PKGS_SAFE_WP_TIMEOUT 61h - MCA_PKGS_CPD_UNCPD_TIMEOUT 62h - MCA_PKGS_INVALID_REQ_PCH 63h - MCA_PKGS_INVALID_REQ_INTERNAL 64h - MCA_PKGS_INVALID_RSP_INTERNAL 65h-7Ah - MCA_PKGS_RESET_PREP_TIMEOUT 7Bh - MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 7Ch - MCA_PKGS_SMBUS_MCP_PAUSE_TIMEOUT 7Dh - MCA_PKGS_SMBUS_SPD_PAUSE_TIMEOUT 80h - MCA_PKGC_DISP_BUSY_TIMEOUT 81h - MCA_PKGC_INVALID_RSP_PCH 83h - MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 84h - MCA_PKGC_WATCHDOG_HANG_CBZ_UP 87h - MCA_PKGC_WATCHDOG_HANG_C2_BLKMASTER 88h - MCA_PKGC_WATCHDOG_HANG_C2_PSLIMIT 89h - MCA_PKGC_WATCHDOG_HANG_SETDISP 8Bh - MCA_PKGC_ALLOW_L1_ERROR 90h - MCA_RECOVERABLE_DIE_THERMAL_TOO_HOT A0h - MCA_ADR_SIGNAL_TIMEOUT A1h - MCA_BCLK_FREQ_OC_ABOVE_THRESHOLD B0h - MCA_DISPATCHER_RUN_BUSY_TIMEOUT
	37:32	ENH_MCA_AVAIL0	Available when Enhanced MCA is in use.
	52:38	CORR_ERR_COUNT	Correctable error count.
	54:53	CORRERRORSTATUSIND	These bits are used to indicate when the number of corrected errors has exceeded the safe threshold to the point where an uncorrected error has become more likely to happen. Table 3 shows the encoding of these bits.
	56:55	ENH_MCA_AVAIL1	Available when Enhanced MCA is in use.
Status register validity indicators ¹	63:57		

NOTES:

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

16.11.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the MSRs IA32_MC5_STATUS, IA32_MC7_STATUS, IA32_MC8_STATUS. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL (see Chapter 15, "Machine-Check Architecture").

NOTE

The interconnect machine check errors in this section apply only to future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6AH. These do not apply to future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6CH.

Table 16-35 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, i= 5, 7, 8.

Table 16-35. Interconnect MC Error Codes for IA32_MCi_STATUS, i = 5, 7, 8

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: - 0x0COF - Unsupported/Undefined Packet. - 0x0EOF - For all other corrected and uncorrected errors.
Model specific errors	21:16	MSCOD	The encoding of Uncorrectable (UC) errors are: 00h - Phy Initialization Failure (NumInit). 01h - Phy Detected Drift Buffer Alarm. 02h - Phy Detected Latency Buffer Rollover. 10h - LL Rx detected CRC error: unsuccessful LLR (entered Abort state). 11h - LL Rx Unsupported/Undefined packet. 12h - LL or Phy Control Error. 13h - LL Rx Parameter Exception. 1Fh - LL Detected Control Error. The encoding of correctable (COR) errors are: 20h - Phy Initialization Abort. 21h - Phy Inband Reset. 22h - Phy Lane failure, recovery in x8 width. 23h - Phy LOc error corrected without Phy reset. 24h - Phy LOc error triggering Phy reset. 25h - Phy LOp exit error corrected with reset. 30h - LL Rx detected CRC error: successful LLR without Phy Reinit. 31h - LL Rx detected CRC error: successful LLR with Phy Reinit. 32h - Tx received LLR. All other values are reserved.

Type	Bit No.	Bit Function	Bit Description
	31:22	MSCOD_SPARE	The definition below applies to MSCOD 12h (UC LL or Phy Control Errors). [Bit 22] : Phy Control Error. [Bit 23] : Unexpected Retry.Ack flit. [Bit 24] : Unexpected Retry.Req flit. [Bit 25] : RF parity error. [Bit 26] : Routeback Table error. [Bit 27] : Unexpected Tx Protocol flit (EOP, Header or Data). [Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow. [Bit 29] : Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTLCL.LinkLayerReset going back to 0). [Bit 30] : Link Layer reset initiated while protocol traffic not idle. [Bit 31] : Link Layer Tx Parity Error.
	37:32	OTHER_INFO	Other Info.
	56:38	Corrected Error Cnt	See Chapter 15, "Machine-Check Architecture".
Status register validity indicators ¹	63:57		

NOTES:

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

16.11.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6AH are reported in the MSRs IA32_MC13_STATUS – IA32_MC15_STATUS, IA32_MC17_STATUS – IA32_MC19_STATUS, IA32_MC21_STATUS – IA32_MC23_STATUS, and IA32_MC25_STATUS – IA32_MC27_STATUS.

MC error codes associated with integrated memory controllers for future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6CH are reported in the MSRs IA32_MC13_STATUS – IA32_MC15_STATUS and IA32_MC17_STATUS – IA32_MC19_STATUS.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture").

Table 16-36. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13-15, 17-19, 21-23, 25-27)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model specific errors	31:16	Reserved except for the following	0001H - Address parity error. 0002H - Data parity error. 0003H - Data ECC error. 0004H - Data byte enable parity error. 0005H - Received uncorrectable data. 0006H - Received uncorrectable metadata. 0007H - Transaction ID parity error. 0008H - Corrected patrol scrub error. 0010H - Uncorrected patrol scrub error. 0020H - Corrected spare error. 0040H - Uncorrected spare error. 0080H - Corrected read error. 00A0H - Uncorrected read error. 00C0H - Uncorrected metadata. 0100H - WDB read parity error. 0103H - RPA parity error. 0104H - RPA parity error. 0105H - WPA parity error. 0106H - DDR_T_DPPP data BE error. 0107H - DDR_T_DPPP data error. 0108H - DDR link failure. 0111H - PCLS CAM error. 0112H - PCLS data error. 0200H - DDR4 command / address parity error. 0220H - HBM command / address parity error. 0221H - HBM data parity error. 0400H - RPQ parity (primary) error. 0401H - RPQ parity (buddy) error. 0404H - WPQ parity (primary) error. 0405H - WPQ parity (buddy) error. 0408H - RPB parity (primary) error. 0409H - RPB parity (buddy) error. 0800H - DDR-T bad request. 0801H - DDR Data response to an invalid entry. 0802H - DDR data response to an entry not expecting data. 0803H - DDR4 completion to an invalid entry. 0804H - DDR-T completion to an invalid entry. 0805H - DDR data/completion FIFO overflow. 0806H - DDR-T ERID correctable parity error. 0807H - DDR-T ERID uncorrectable error. 0808H - DDR-T interrupt received while outstanding interrupt was not ACKed.

INTERPRETING MACHINE-CHECK ERROR CODES

Type	Bit No.	Bit Function	Bit Description
			0809H - ERID FIFO overflow. 080AH - DDR-T error on FNV write credits. 080BH - DDR-T error on FNV read credits. 080CH - DDR-T scheduler error. 080DH - DDR-T FNV error event. 080EH - DDR-T FNV thermal event. 080FH - CMI packet while idle. 0810H - DDR_T_RPQ_REQ_PARITY_ERR. 0811H - DDR_T_WPQ_REQ_PARITY_ERR. 0812H - 2LM_NMFILLWR_CAM_ERR. 0813H - CMI_CREDIT_OVERSUB_ERR. 0814H - CMI_CREDIT_TOTAL_ERR. 0815H - CMI_CREDIT_RSVD_POOL_ERR. 0816H - DDR_T_RD_ERROR. 0817H - WDB_FIFO_ERR. 0818H - CMI_REQ_FIFO_OVERFLOW. 0819H - CMI_REQ_FIFO_UNDERFLOW. 081AH - CMI_RSP_FIFO_OVERFLOW. 081BH - CMI_RSP_FIFO_UNDERFLOW. 081CH - CMI_MISC_MC_CRDT_ERRORS. 081DH - CMI_MISC_MC_ARB_ERRORS. 081EH - DDR_T_WR_CMPL_FIFO_OVERFLOW. 081FH - DDR_T_WR_CMPL_FIFO_UNDERFLOW. 0820H - CMI_RD_CPL_FIFO_OVERFLOW. 0821H - CMI_RD_CPL_FIFO_UNDERFLOW. 0822H - TME_KEY_PAR_ERR. 0823H - TME_CMI_MISC_ERR. 0824H - TME_CMI_OVFL_ERR. 0825H - TME_CMI_UFL_ERR. 0826H - TME_TEM_SECURE_ERR. 0827H - TME_UFILL_PAR_ERR.
	37:32	Other info	Other Info.
	56:38		See Chapter 15, "Machine-Check Architecture".
Status register validity indicators ¹	63:57		

NOTES:

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

Additional information is reported in the MSRs IA32_MC13_MISC – IA32_MC15_MISC, IA32_MC17_MISC – IA32_MC19_MISC, IA32_MC21_MISC – IA32_MC23_MISC, and IA32_MC25_MISC – IA32_MC27_MISC. Table 16-37 lists the information reported in IA32_MCi_MISC, i = 13-15, 17-19, 21-23, and 25-27.

Table 16-37. Additional Information Reported in IA32_MCi_MISC (i= 13-15, 17-19, 21-23, 25-27)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 15-8.
8:6	Address Mode	See Table 15-3.
18:9	Column	Component of sub-DIMM address. Bits 18-17: Reserved Bit 16: Column 9 Bit 15: Column 8 Bit 14: Column 7 Bit 13: Column 6 Bit 12: Column 5 Bit 11: Column 4 Bit 10: Column 3 Bit 9: Reserved
39:19	Row	Component of sub-DIMM address.
45:40	Bank	Component of sub-DIMM address. Bit 45: Reserved Bit 44: Bank group 2 Bit 43: Bank address 1 Bit 42: Bank address 0 Bit 41: Bank group 1 Bit 40: Bank address 0
51:46	Failed Device	Failing device for correctable error (not valid for uncorrectable or transient errors).
55:52	SubRank	Logical Rank (3D Stacked SDRAM)
58:56	Rank	Rank
62:59	ECC Mode	0000b: SDDC memory mode 0001b: SDDC 0100b: ADDDC memory mode 0101b: ADDDC 1000b: Read from DDRT Other values: Reserved
63	Transient	0b: 1b: Error was transient

16.11.4 M2M Machine Check Errors

MC error codes associated with M2M for future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6AH are reported in the MSRs IA32_MC12_STATUS, IA32_MC16_STATUS, IA32_MC20_STATUS, and IA32_MC24_STATUS.

MC error codes associated with M2M for future Intel Xeon processors with a CPUID DisplayFamily_DisplaySignature of 06_6AH are reported in the MSRs IA32_MC12_STATUS and IA32_MC16_STATUS.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

Table 16-38. M2M MC Error Codes for IA32_MCi_STATUS (i= 12, 16, 20, 24)

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model specific errors	16	MscodDataRdErr	Logged an MC read data error.
	17	Reserved	Reserved
	18	MscodPtlWrErr	Logged an MC partial write data error.
	19	MscodFullWrErr	Logged a full write data error.
	20	MscodBgfErr	Logged an M2M clock-domain-crossing buffer (BGF) error.
	21	MscodTimeOut	Logged an M2M time out.
	22	MscodParErr	Logged an M2M tracker parity error.
	23	MscodBucket1Err	Logged a fatal Bucket1 error.
	25:24	MscodDDRTYPE	Logged a DDR/DDR-T specific error.
	31:26	MscodMiscErrs	Logged a miscellaneous error.
37:32	Other info	Other Info.	
56:38		See Chapter 15, “Machine-Check Architecture”.	
Status register validity indicators ¹	63:57		

NOTES:

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

MC error codes associated with mirrored memory corrections are reported in the MSRs IA32_MC12_MISC, IA32_MC16_MISC, IA32_MC20_MISC, and IA32_MC24_MISC. The model-specific error codes listed in Table 16-32 also apply to IA32_MCi_MISC, i = 12, 16, 20, 24.

16.12 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_86H, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel® Atom® processors based on Tremont microarchitecture with CPUID DisplayFamily_DisplaySignature 06_86H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 16-34 in Section 16.11.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

16.12.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC13_STATUS - IA32_MC15_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

- The IA32_MCi_STATUS MSR (where i = 13, 14, 15) contains information related to a machine check error if its VAL(valid) flag is set. Bit definitions are the same as those found in Table 16-36 “Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13-15, 17-19, 21-23, 25-27)”.

The IA32_MCi_MISC MSR (where i = 13, 14, 15) contains information related memory corrections. Bit definitions are the same as those found in Table 16-37 “Additional Information Reported in IA32_MCi_MISC (i= 13-15, 17-19, 21-23, 25-27)”.

16.12.2 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32_MC12_STATUS MSR. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, “Machine-Check Architecture”).

Bit definitions are the same as those found in Table 16-38 “M2M MC Error Codes for IA32_MCi_STATUS (i= 12, 16, 20, 24)”.

16.13 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH MACHINE ERROR CODES FOR MACHINE CHECK

Table 16-39 provides information for interpreting additional family 0FH model-specific fields for external bus errors. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of 0000 1PPT RRRR IILL in the MCA error code field. See Chapter 15 for information on the interpretation of compound error codes.

Table 16-39. Incremental Decoding Information: Processor Family 0FH Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0		
Model-specific error codes	16	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	17	Response hard fail	Hardware failure detected on response
	18	Response parity	Parity error detected on response
	19	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	20	Processor Signature = 00000F04H: Invalid PIC request All other processors: Reserved	Processor Signature = 00000F04H. Indicates error due to an invalid PIC request access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No Invalid PIC request error Reserved
	21	Pad state machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad strobe glitch	Data strobe glitch
	23	Pad address glitch	Address strobe glitch
Other Information	56:24	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

Table 16-10 provides information on interpreting additional family 0FH, model specific fields for cache hierarchy errors. These errors are reported in one of the IA32_MCi_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of 0000 0001 RRRR TTLL in the MCA error code field. See Chapter 15 for how to interpret the compound error code.

16.13.1 Model-Specific Machine Check Error Codes for Intel Xeon Processor MP 7100 Series

Intel Xeon processor MP 7100 series has 5 register banks which contains information related to Machine Check Errors. MCi_STATUS[63:0] refers to all 5 register banks. MC0_STATUS[63:0] through MC3_STATUS[63:0] is the same as on previous generation of Intel Xeon processors within Family 0FH. MC4_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

Table 16-40. MCI_STATUS Register Bit Definition

Bit Field Name	Bits	Description
MCA_Error_Code	15:0	Specifies the machine check architecture defined error code for the machine check error condition detected. The machine check architecture defined error codes are guaranteed to be the same for all Intel Architecture processors that implement the machine check architecture. See tables below
Model_Specific_Error_Code	31:16	Specifies the model specific error code that uniquely identifies the machine check error condition detected. The model specific error codes may differ among Intel Architecture processors for the same Machine Check Error condition. See tables below
Other_Info	56:32	The functions of the bits in this field are implementation specific and are not part of the machine check architecture. Software that is intended to be portable among Intel Architecture processors should not rely on the values in this field.
PCC	57	Processor Context Corrupt flag indicates that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state. This bit will always be set for MC errors which are not corrected.
ADDRV	58	MC_ADDR register valid flag indicates that the MC_ADDR register contains the address where the error occurred. When clear, this flag indicates that the MC_ADDR register does not contain the address where the error occurred. The MC_ADDR register should not be read if the ADDRv bit is clear.
MISCV	59	MC_MISC register valid flag indicates that the MC_MISC register contains additional information regarding the error. When clear, this flag indicates that the MC_MISC register does not contain additional information regarding the error. MC_MISC should not be read if the MISCV bit is not set.
EN	60	Error enabled flag indicates that reporting of the machine check exception for this error was enabled by the associated flag bit of the MC_CTL register. Note that correctable errors do not have associated enable bits in the MC_CTL register so the EN bit should be clear when a correctable error is logged.

Table 16-40. MCI_STATUS Register Bit Definition (Contd.)

Bit Field Name	Bits	Description
UC	61	Error uncorrected flag indicates that the processor did not correct the error condition. When clear, this flag indicates that the processor was able to correct the event condition.
OVER	62	Machine check overflow flag indicates that a machine check error occurred while the results of a previous error were still in the register bank (i.e., the VAL bit was already set in the MC_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. Enabled errors are written over disabled errors, and uncorrected errors are written over corrected events. Uncorrected errors are not written over previous valid uncorrected errors.
VAL	63	MC_STATUS register valid flag indicates that the information within the MC_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the MC_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

16.13.1.1 Processor Machine Check Status Register MCA Error Code Definition

Intel Xeon processor MP 7100 series use compound MCA Error Codes for logging its CBC internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines additional Machine Check error types (IA32_MC4_STATUS[15:0]) beyond those defined in Chapter 15. Table 16-41 lists these model-specific MCA error codes. Error code details are specified in MC4_STATUS [31:16] (see Section 16.13.3), the "Model Specific Error Code" field. The information in the "Other_Info" field (MC4_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC4_MISC register format.

Table 16-41. Incremental MCA Error Code for Intel Xeon Processor MP 7100

Processor MCA_Error_Code (MC4_STATUS[15:0])			
Type	Error Code	Binary Encoding	Meaning
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code
A	L3 Tag Error	0000 0001 0000 1011	L3 Tag Error Type Code
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 101x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 110x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations
		0000 1110 0000 1111	Bus and Interconnection Error Type Code
		0000 1111 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations

The **Bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

16.13.2 Other_Info Field (all MCA Error Types)

The MC4_STATUS[56:32] field is common to the processor's three MCA error types (A, B & C).

Table 16-42. Other Information Field Bit Definition

Bit Field Name	Bits	Description
39:32	8-bit Correctable Event Count	Holds a count of the number of correctable events since cold reset. This is a saturating counter; the counter begins at 1 (with the first error) and saturates at a count of 255.
41:40	MC4_MISC format type	The value in this field specifies the format of information in the MC4_MISC register. Currently, only two values are defined. Valid only when MISCV is asserted.
43:42	-	Reserved
51:44	ECC syndrome	ECC syndrome value for a correctable ECC event when the "Valid ECC syndrome" bit is asserted
52	Valid ECC syndrome	Set when correctable ECC event supplies the ECC syndrome
54:53	Threshold-Based Error Status	00: No tracking - No hardware status tracking is provided for the structure reporting this event. 01: Green - Status tracking is provided for the structure posting the event; the current status is green (below threshold). 10: Yellow - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). 11: Reserved for future use Valid only if Valid bit (bit 63) is set Undefined if the UC bit (bit 61) is set
56:55	-	Reserved

16.13.3 Processor Model Specific Error Code Field

16.13.3.1 MCA Error Type A: L3 Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

Table 16-43. Type A: L3 Error Codes

Bit Num	Sub-Field Name	Description	Legal Value(s)
18:16	L3 Error Code	Describes the L3 error encountered	000 - No error 001 - More than one way reporting a correctable event 010 - More than one way reporting an uncorrectable error 011 - More than one way reporting a tag hit 100 - No error 101 - One way reporting a correctable event 110 - One way reporting an uncorrectable error 111 - One or more ways reporting a correctable event while one or more ways are reporting an uncorrectable error
20:19	-	Reserved	00
31:21	-	Fixed pattern	0010_0000_000

16.13.3.2 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

Table 16-44. Type B Bus and Interconnect Error Codes

Bit Num	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase
17	Core0 Addr Parity	Parity error detected on Core 0 request's address field
18	Core1 Addr Parity	Parity error detected on Core 1 request's address field
19		Reserved
20	FSB Response Parity	Parity error on FSB response field detected
21	FSB Data Parity	FSB data parity error on inbound data detected
22	Core0 Data Parity	Data parity error on data received from Core 0 detected
23	Core1 Data Parity	Data parity error on data received from Core 1 detected
24	IDS Parity	Detected an Enhanced Defer parity error (phase A or phase B)
25	FSB Inbound Data ECC	Data ECC event to error on inbound data (correctable or uncorrectable)
26	FSB Data Glitch	Pad logic detected a data strobe 'glitch' (or sequencing error)
27	FSB Address Glitch	Pad logic detected a request strobe 'glitch' (or sequencing error)
31:28	---	Reserved

Exactly one of the bits defined in the preceding table will be set for a Bus and Interconnect Error. The Data ECC can be correctable or uncorrectable (the MC4_STATUS.UC bit, of course, distinguishes between correctable and uncorrectable cases with the Other_Info field possibly providing the ECC Syndrome for correctable errors). All other errors for this processor MCA Error Type are uncorrectable.

16.13.3.3 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

Table 16-45. Type C Cache Bus Controller Error Codes

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0
0000_0000_0000_0010 0002H	Inclusion Error from Core 1
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1
0000_0000_0000_0101 0005H	Inclusion Error from FSB
0000_0000_0000_0110 0006H	SNP Stall Error from FSB
0000_0000_0000_0111 0007H	Write Stall Error from FSB
0000_0000_0000_1000 0008H	FSB Arb Timeout Error
0000_0000_0000_1001 0009H	CBC OOD Queue Underflow/overflow
0000_0001_0000_0000 0100H	Enhanced Intel SpeedStep Technology TM1-TM2 Error
0000_0010_0000_0000 0200H	Internal Timeout error
0000_0011_0000_0000 0300H	Internal Timeout Error
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow

Table 16-45. Type C Cache Bus Controller Error Codes (Contd.)

MC4_STATUS[31:16] (MSCE) Value	Error Description
1100_0000_0000_0001 C001H	Correctable ECC event on outgoing FSB data
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data
1110_0000_0000_0001 E001H	Uncorrectable ECC error on outgoing FSB data
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data
— all other encodings —	Reserved

All errors - except for the correctable ECC types - in this table are uncorrectable. The correctable ECC events may supply the ECC syndrome in the Other_Info field of the MC4_STATUS MSR.

Table 16-46. Decoding Family 0FH Machine Check Codes for Cache Hierarchy Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes ¹	15:0		
Model specific error codes	17:16	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	19:18	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	31:22	Reserved	Reserved
Other Information	39:32	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 255.
	56:40	Reserved	Reserved
Status register validity indicators ¹	63:57		

NOTES:

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

24. Updates to Chapter 17, Volume 3B

Change bars and green text 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 chapter: Update to Figure 17-2 "DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture".

CHAPTER 17

DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) 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 17.17, “Time-Stamp Counter”.
- Features which allow monitoring of shared platform resources such as the L3 cache are described in Section 17.18, “Intel® Resource Director Technology (Intel® RDT) Monitoring Features”.
- Features which enable control over shared platform resources are described in Section 17.19, “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 (INT3) 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 (INT3)** — Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set instruction 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.

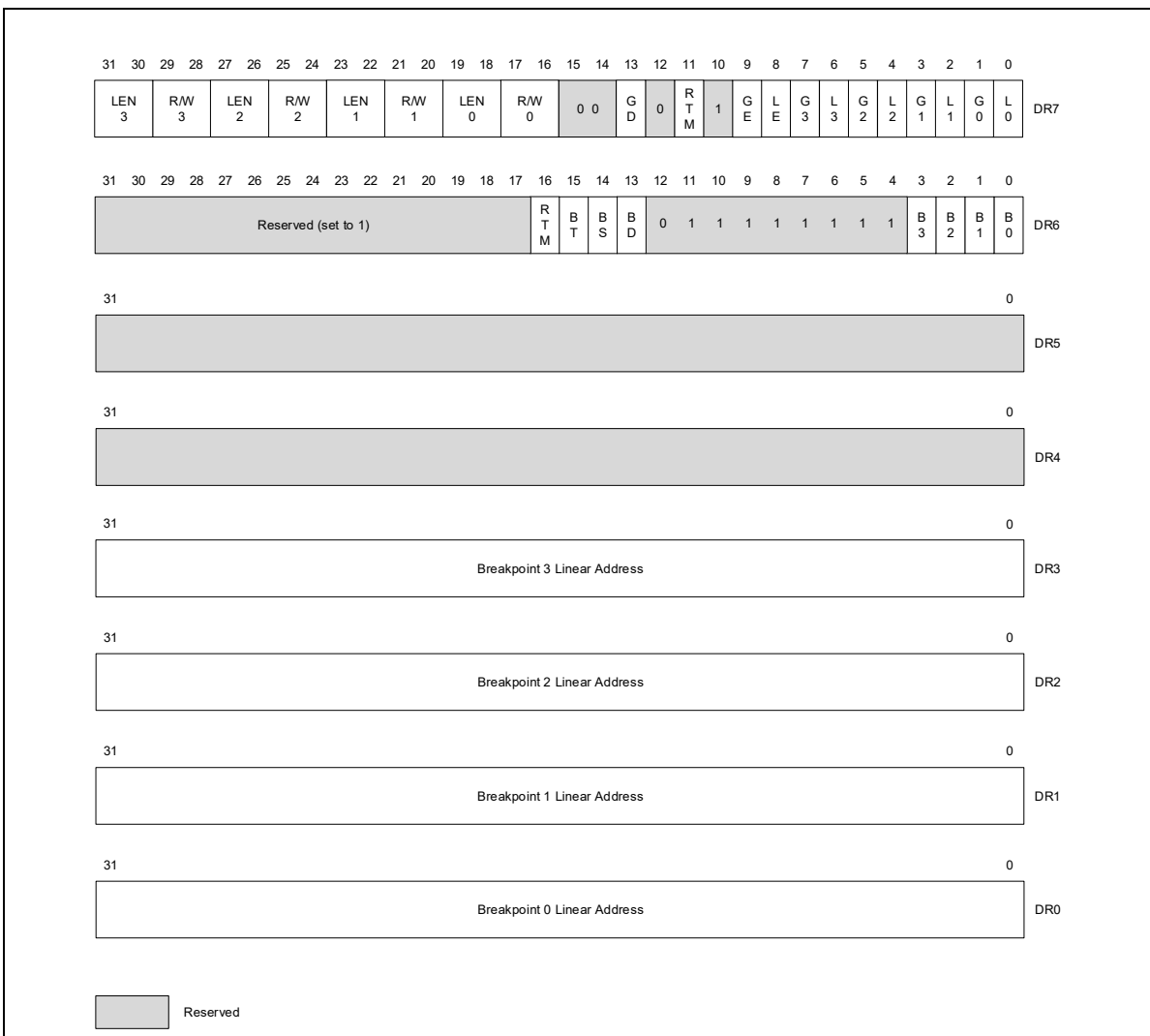


Figure 17-1. Debug Registers

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 through 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 RWn field in register DR7 is 00 (instruction execution), then the $LENn$ 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), breakpoint conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the $LENn$ 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 $LENn$ fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The $LENn$ fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DRn). 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 $LENn$ 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 $LENn$ 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 $LENn$ field is set to 00). Instruction 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/Wn	Breakpoint Address	LENn
DR0	R/W0 = 11 (Read/Write)	A0001H	LEN0 = 00 (1 byte)
DR1	R/W1 = 01 (Write)	A0002H	LEN1 = 00 (1 byte)
DR2	R/W2 = 11 (Read/Write)	B0002H	LEN2 = 01) (2 bytes)
DR3	R/W3 = 01 (Write)	C0000H	LEN3 = 11 (4 bytes)
Data Accesses			
Operation		Address	Access Length (In Bytes)
Data operations that trap			
- Read or write		A0001H	1
- Read or write		A0001H	2
- Write		A0002H	1
- Write		A0002H	2
- Read or write		B0001H	4
- Read or write		B0002H	1
- Read or write		B0002H	2
- Write		C0000H	4
- Write		C0001H	2
- Write		C0003H	1
Data operations that do not trap			
- Read or write		A0000H	1
- Read		A0002H	1
- Read or write		A0003H	4
- Read or write		B0000H	2
- Read		C0000H	2
- Read or write		C0004H	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.

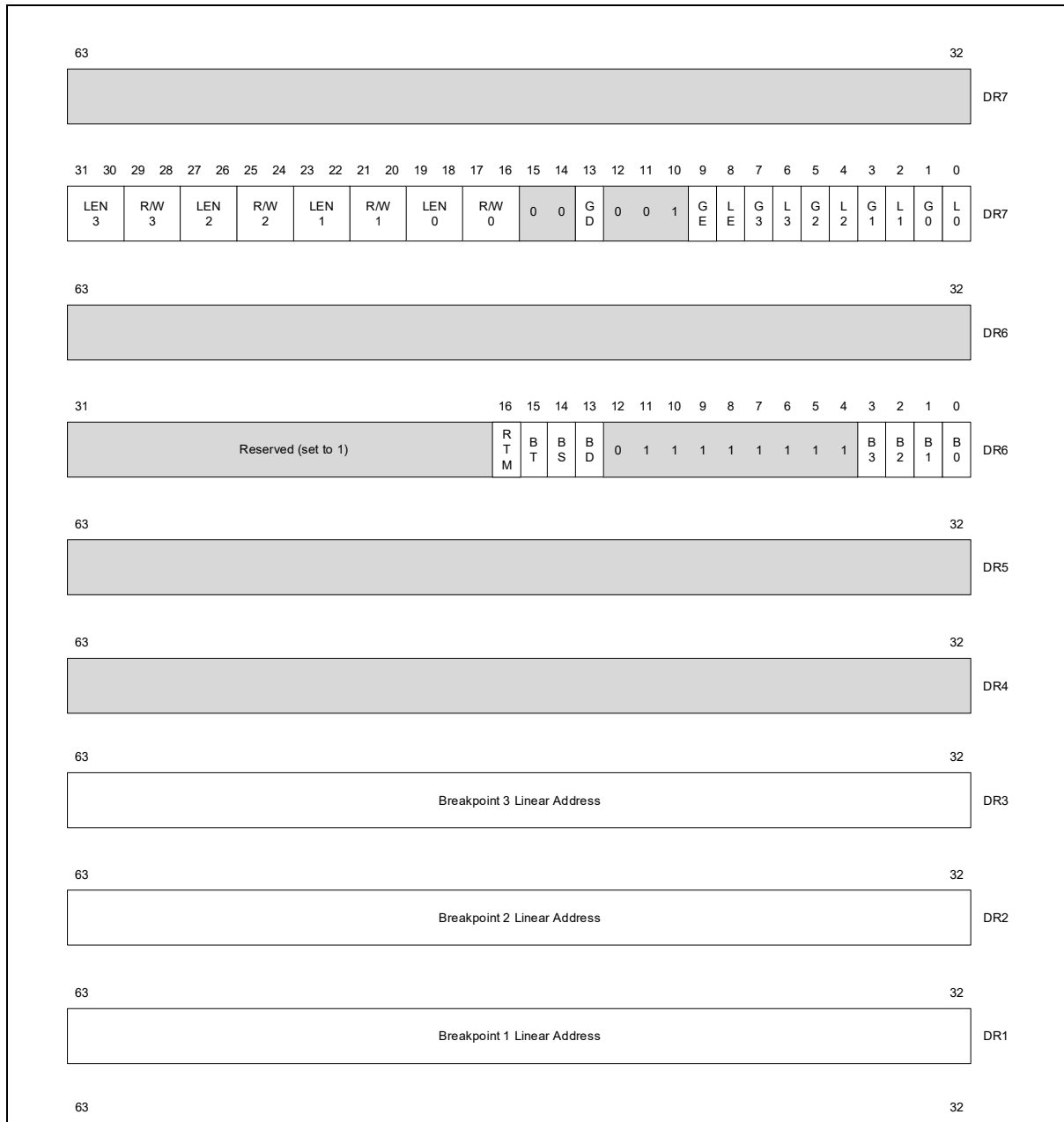


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

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.

The INT1 instruction generates a debug exception as a trap. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

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	None	Fault
Task switch	BT = 1	None	Trap
INT1 instruction	None	None	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 an instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint will be suppressed as if EFLAGS.RF were 1 (see the next paragraph and Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

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 instruction breakpoints, CS limit violations, 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.

If an execution of the MOV or POP instruction loads the SS register and encounters a data breakpoint, the resulting debug exception is delivered after completion of the next instruction (the one after the MOV or POP).

Any pending data or I/O breakpoints are lost upon delivery of an exception. For example, if a machine-check exception (#MC) occurs following an instruction that encounters a data breakpoint (but before the resulting debug exception is delivered), the data breakpoint is lost. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, the data breakpoint is lost if the next instruction causes a fault.

Delivery of events due to INT *n*, INT3, or INTO does not cause a loss of data breakpoints. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, and the next instruction is software interrupt (INT *n*, INT3, or INTO), a debug exception (#DB) resulting from a data breakpoint will be delivered after the transition to the software-interrupt handler. The #DB handler should account for the fact that the #DB may have been delivered

after a invocation of a software-interrupt handler, and in particular that the CPL may have changed between recognition of the data breakpoint and delivery of the #DB.

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*, INT3, 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.

If an occurrence of the MOV or POP instruction loads the SS register executes with EFLAGS.TF = 1, no single-step debug exception occurs following the MOV or POP instruction.

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 INT3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP)." Debuggers use breakpoint 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 17.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem”
- Section 17.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”
- Section 17.11, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture”
- Section 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture”
- Section 17.14, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
- Section 17.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)”
- Section 17.16, “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 17.9.1, “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.

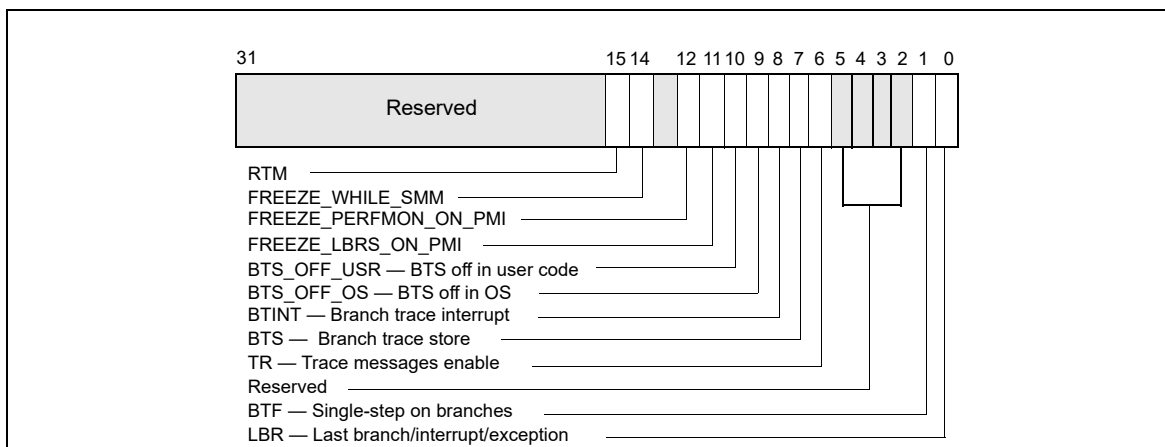


Figure 17-3. IA32_DEBUGCTL MSR for Processors based on Intel Core microarchitecture

- **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 17.13.2.
- **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 17.13.2.
- **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 (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 control bit.

IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.8 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).

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).”

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.

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 18.6.2.1, “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, 06_55H, 06_66H, 06_7AH, 06_67H, 06_6AH, 06_6CH, 06_7DH, 06_7EH	32	FROM_IP, TO_IP, LBR_INFO ¹	0 to 31
06_3DH, 06_47H, 06_4FH, 06_56H, 06_3CH, 06_45H, 06_46H, 06_3FH, 06_2AH, 06_2DH, 06_3AH, 06_3EH, 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, 06_0FH	4	FROM_IP, TO_IP	0 to 3
06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH, 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	8	FROM_IP, TO_IP	0 to 7

NOTES:

1. See Section 17.12.

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.

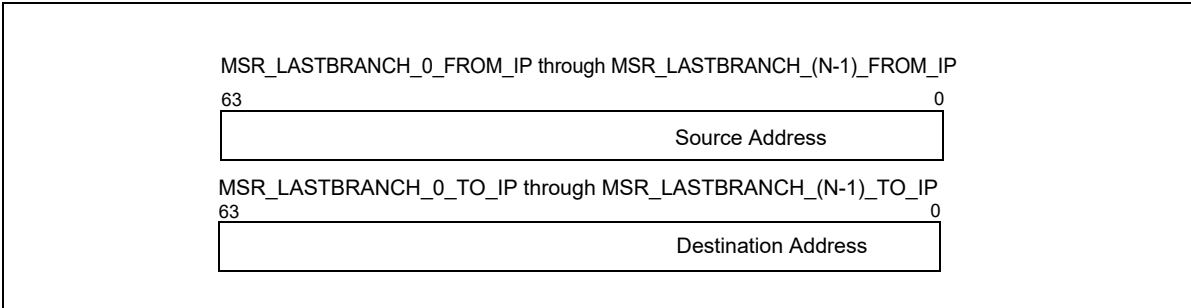


Figure 17-4. 64-bit Address Layout of LBR MSR

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 LIP 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).
- **000111B (64-bit LIP record format), Flags, LBR_INFO** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.

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 18.6.2.4, “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 18.6.2.4.2 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 18.5.3.1 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: buffer management area, branch trace store (BTS) buffer, and PEBS buffer (see Figure 17-5). 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 64-bit value that the counter is to be set to when a PEBS record is written. Bits beyond the size of the counter are ignored. This value allows state information to be collected regularly every time the specified number of events occur.

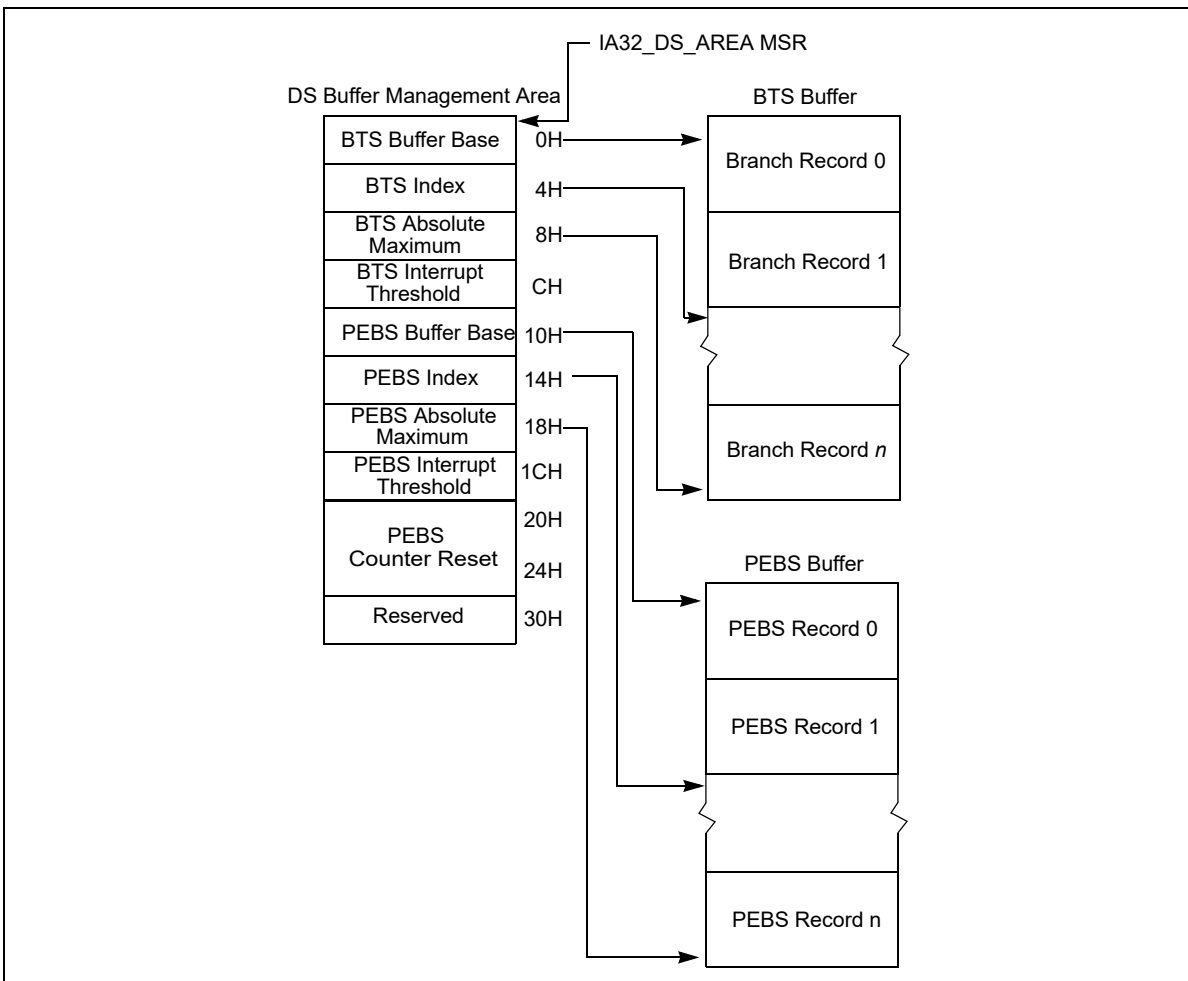


Figure 17-5. DS Save Area Example¹

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

Figure 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).

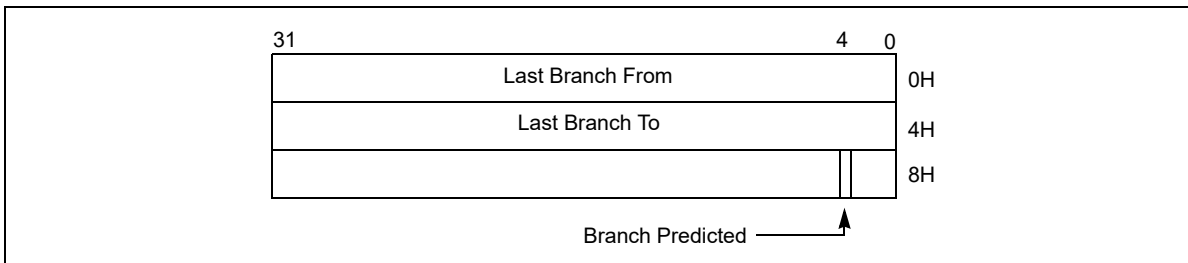


Figure 17-6. 32-bit Branch Trace Record Format

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.

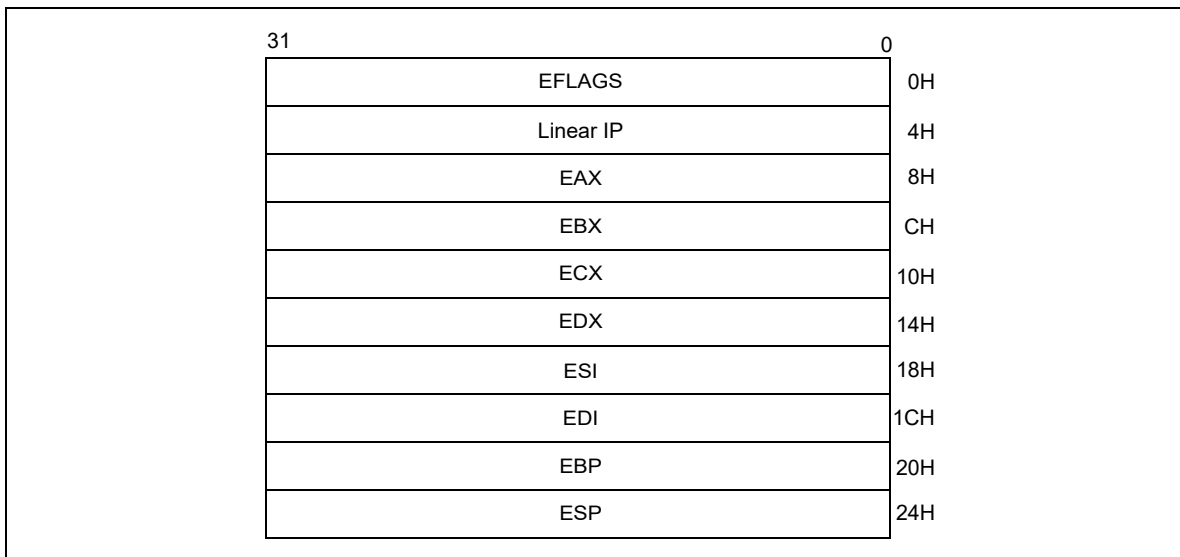


Figure 17-7. PEBS Record Format

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 17-5.

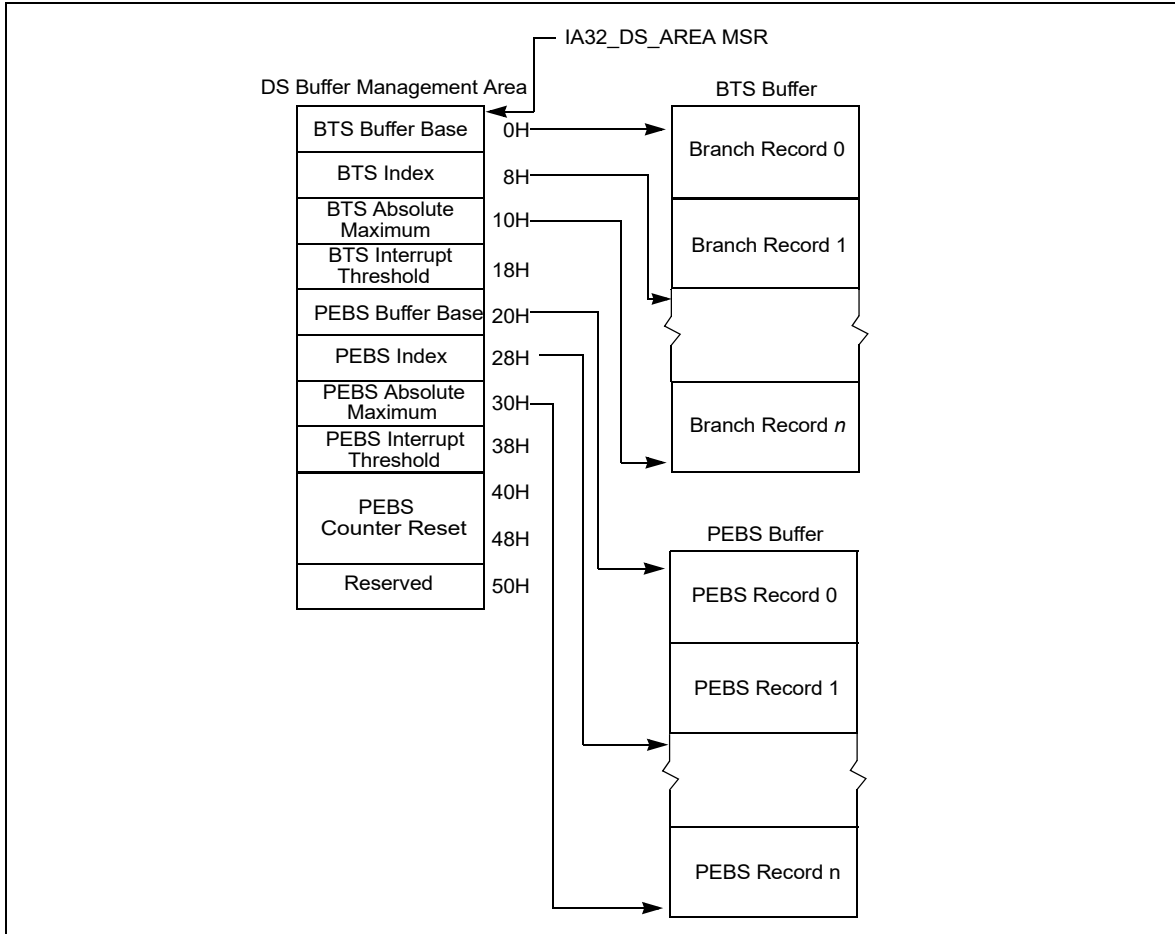


Figure 17-8. IA-32e Mode DS Save Area Example¹

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

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).

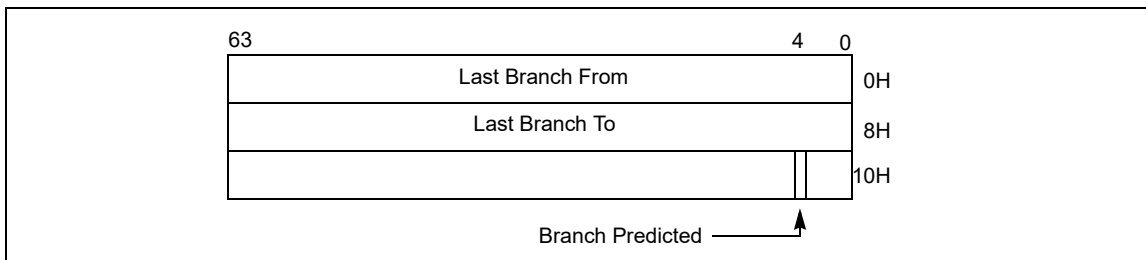


Figure 17-9. 64-bit Branch Trace Record Format

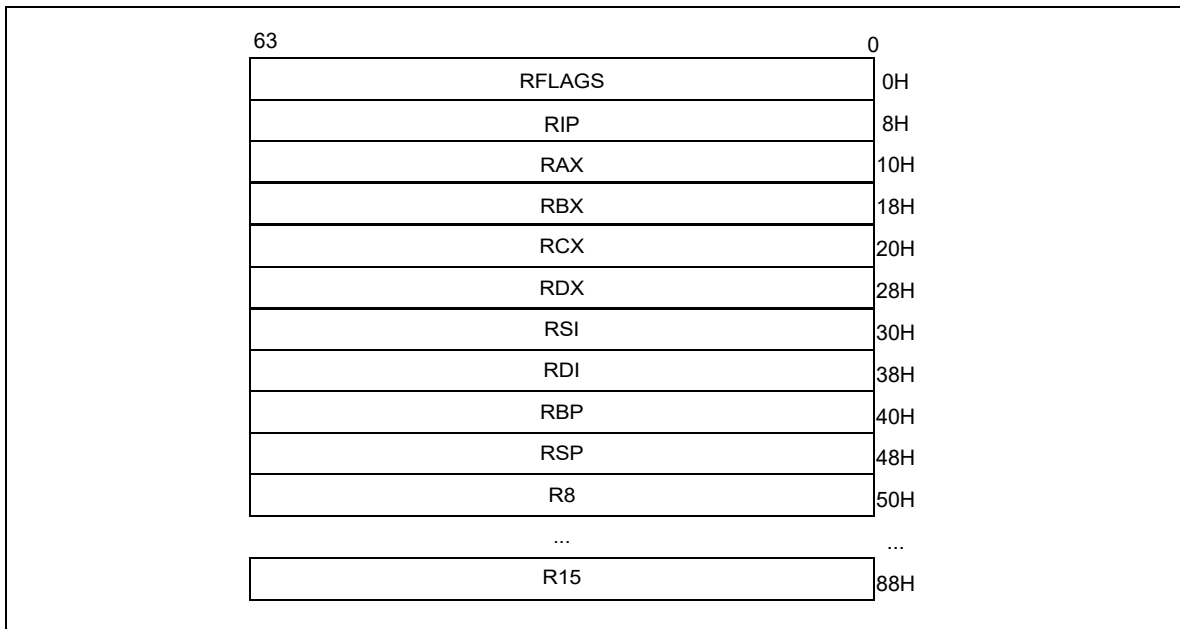


Figure 17-10. 64-bit PEBS Record Format

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 18.6.2.4.1, “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 be 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 17.13.3.
- **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 (bit 14)** — FREEZE_WHILE_SMM 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

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. MISRPRED bit format is shown in Table 17-8.
- Streamlined Freeze_LBRs_On_PMI operation; see Section 17.12.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 17.12.3.

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, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 17.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of three 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. Layout is the same as Table 17-16.

17.8 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.

17.9 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 17.9.1.

- **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 17.13.3.
- **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 (bit 14)** — FREEZE_WHILE_SMM 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.

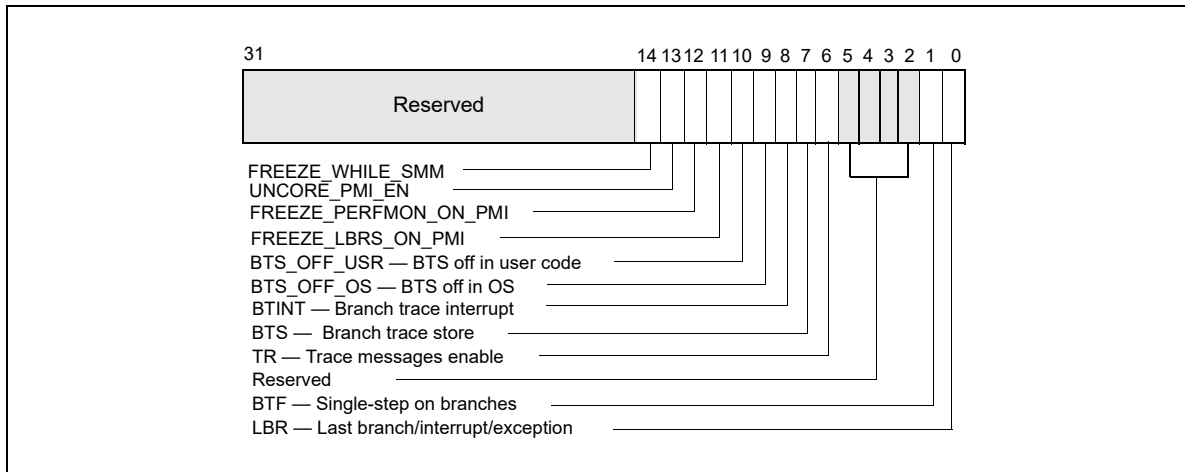


Figure 17-11. IA32_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem

17.9.1 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

17.9.2 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 ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending 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

17.10 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 17.9, “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 ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending 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
Reserved	63:9		Must be zero

17.11 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 17.10, “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 17.10.

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 ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending 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.

Table 17-13. MSR_LBR_SELECT for Intel® microarchitecture code name Haswell

Bit Field	Bit Offset	Access	Description
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.)

17.11.1 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

Table 17-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information (Contd.)

Bit Field	Bit Offset	Access	Description
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.

17.12 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

17.12.1 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.

17.12.2 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.

17.12.3 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.

17.13 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 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- **Last exception record** — See Section 17.13.3, “Last Exception Records.”

17.13.1 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 17.13.2, “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.”

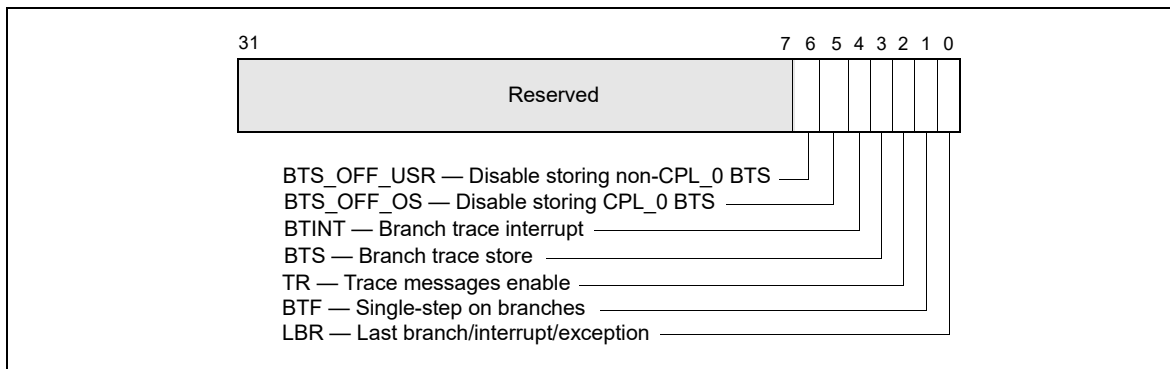


Figure 17-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **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 17.13.2, “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 17.13.2, “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.

17.13.2 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 its 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 is 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.

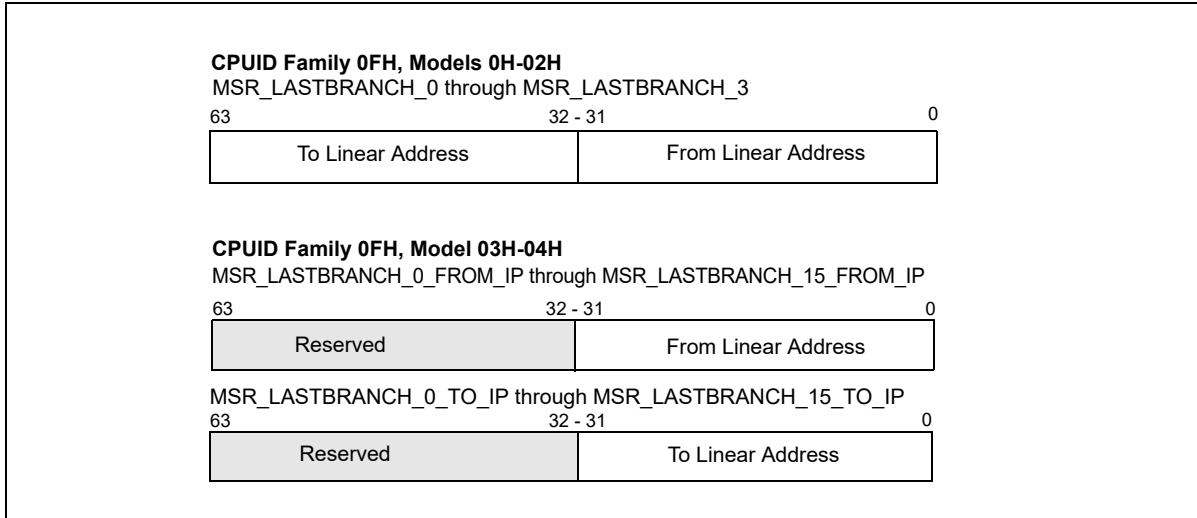


Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

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.

17.13.3 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.

17.14 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.

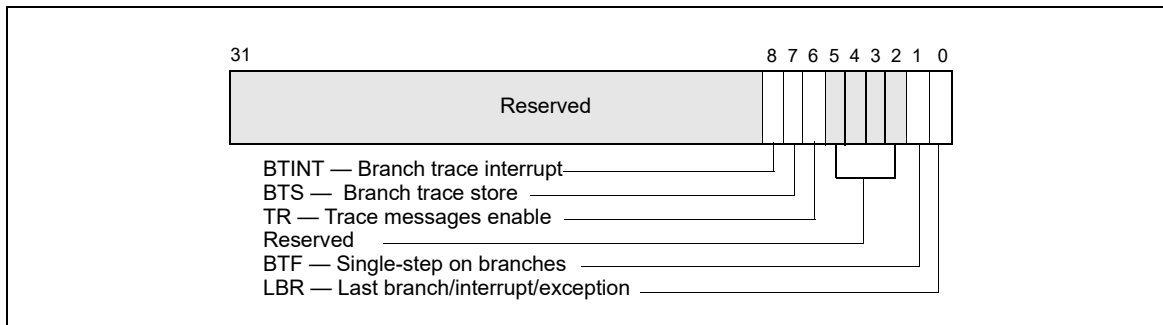


Figure 17-14. IA32_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors

- **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 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture,” and Section 2.20, “MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

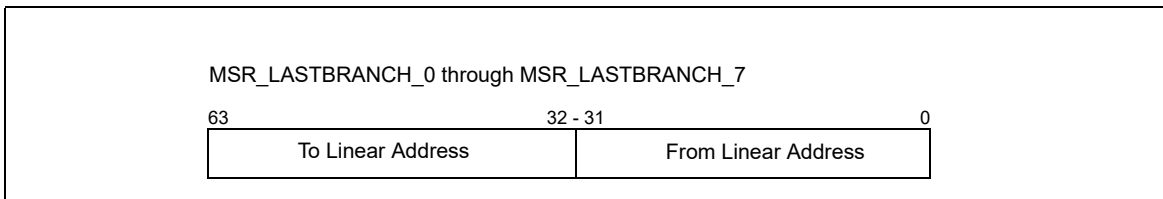


Figure 17-15. LBR Branch Record Layout for the Intel Core Solo and Intel Core Duo Processor

17.15 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.
 - **PBi (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 BPi# pin when a breakpoint match occurs. When a PBi 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.

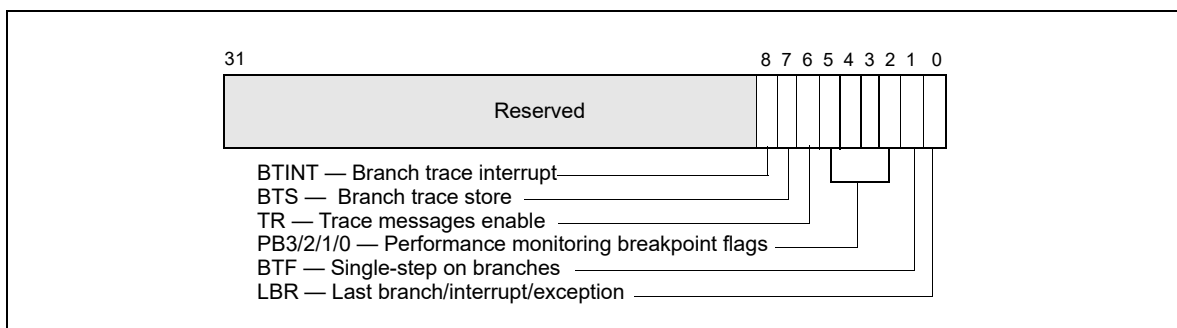


Figure 17-16. MSR_DEBUGCTLB MSR for Pentium M Processors

- **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 MSR (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.

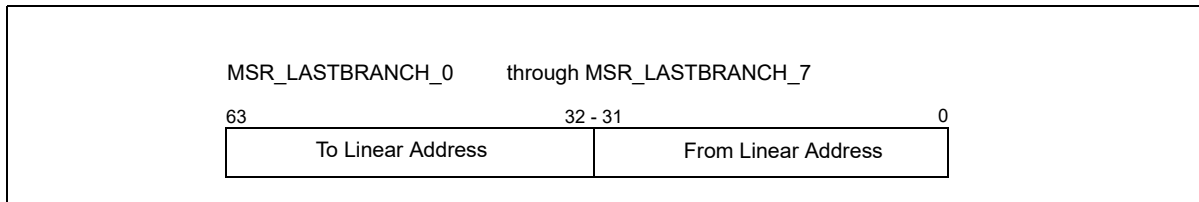


Figure 17-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 17.13.3, “Last Exception Records,” and Section 2.21, “MSRs In the Pentium M Processor” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

17.16 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.

17.16.1 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.

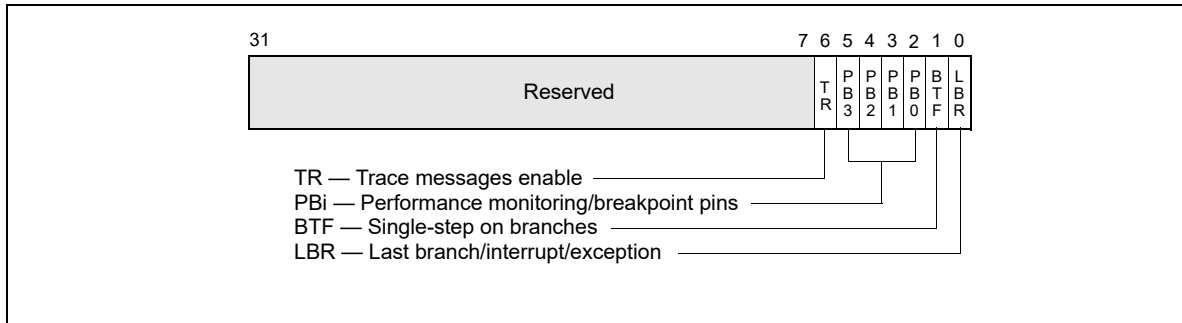


Figure 17-18. DEBUGCTLMR Register (P6 Family Processors)

- **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.”
- **PBi (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 BPi# pin when a breakpoint match occurs. When a PBi 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.

17.16.2 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.

17.16.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMR 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.

17.17 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 18.7.2 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 18.6.4.5, “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.

17.17.1 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.

17.17.2 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.

17.17.3 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 that 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).

17.17.4 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:

$$\text{TSC_Value} = (\text{ART_Value} * \text{CPUID.15H:EBX[31:0]}) / \text{CPUID.15H:EAX[31:0]} + K$$

Where 'K' is an offset that can be adjusted by a privileged agent².

When ART hardware is reset, both invariant TSC and K are also reset.

17.18 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

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

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.

17.18.1 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).

17.18.2 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.

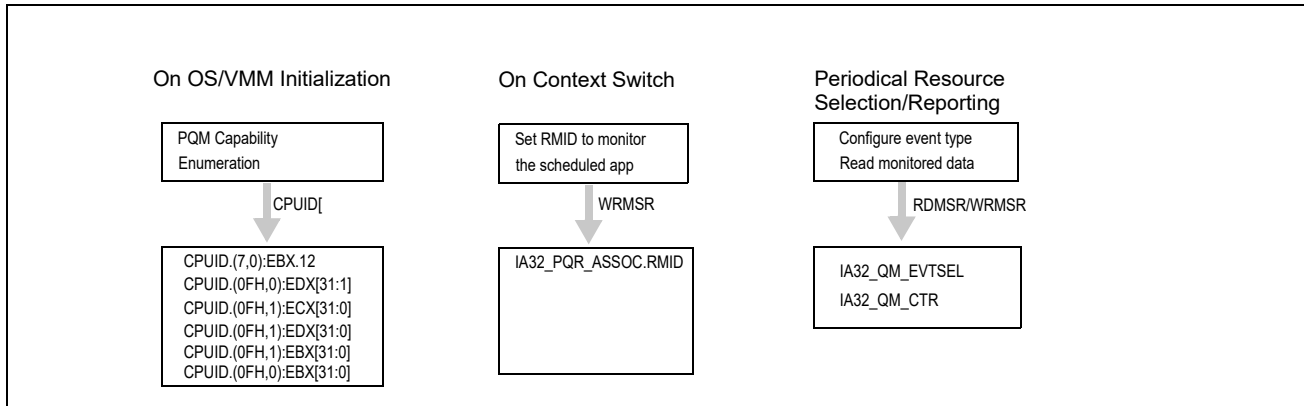


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

17.18.3 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 17.18.4), and monitoring capabilities for each resource type (see Section 17.18.5). 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 17.18.6.
- 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 17.18.7.
- 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.

17.18.4 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.

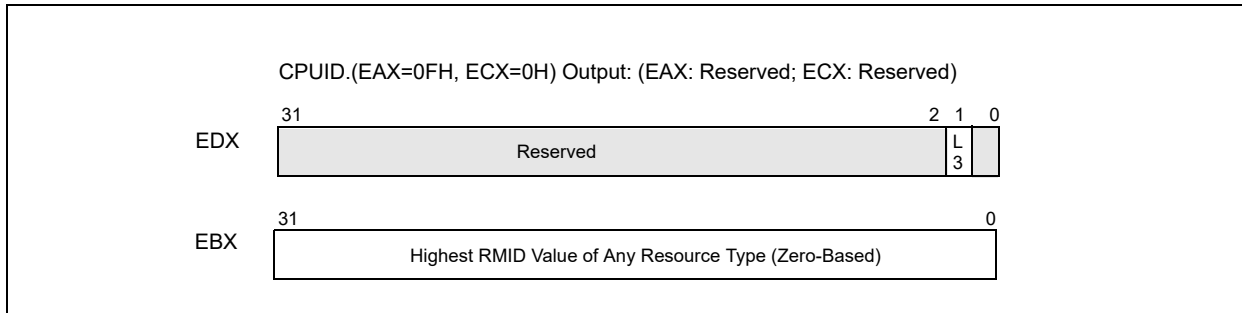


Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

17.18.5 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.

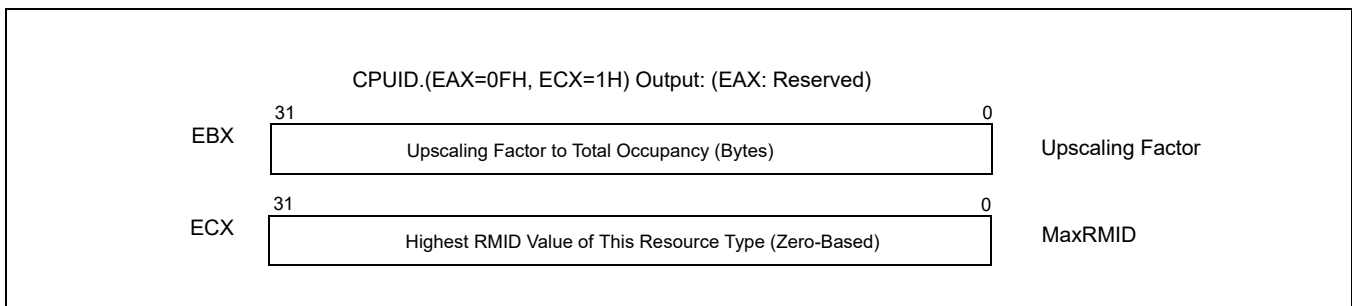


Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))

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.

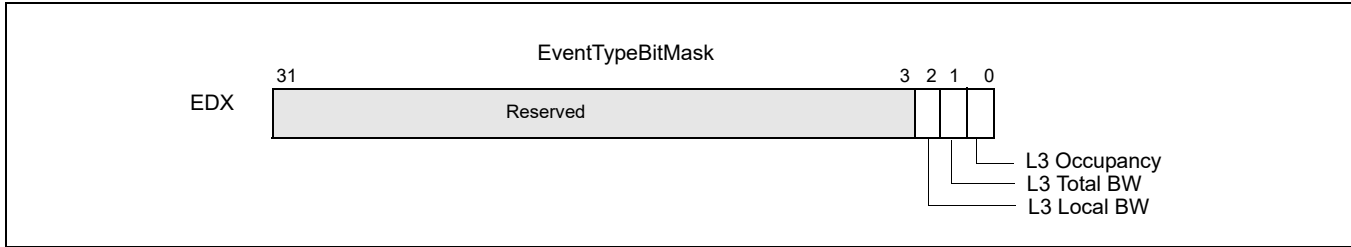


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))

17.18.5.1 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.

17.18.5.2 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

17.18.6 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 ($\log_2 (1 + \text{CPUID}.\text{EAX}=\text{0FH}, \text{ECX}=\text{0}): \text{EBX}[31:0])$). The value of IA32_PQR_ASSOC after power-on is 0.

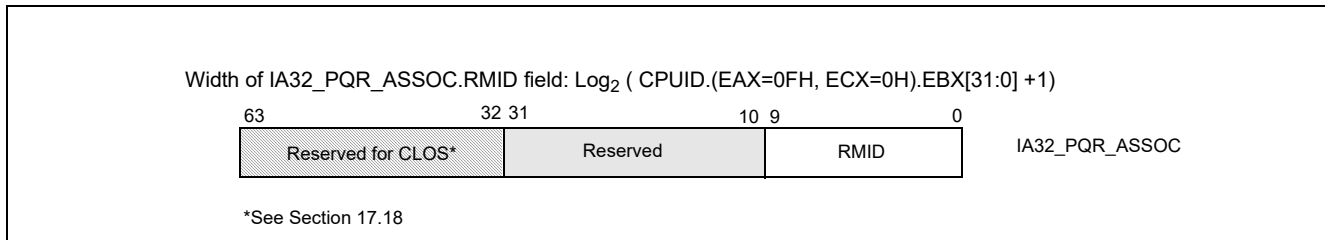


Figure 17-23. IA32_PQR_ASSOC MSR

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.

17.18.7 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 17.18.5, 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.

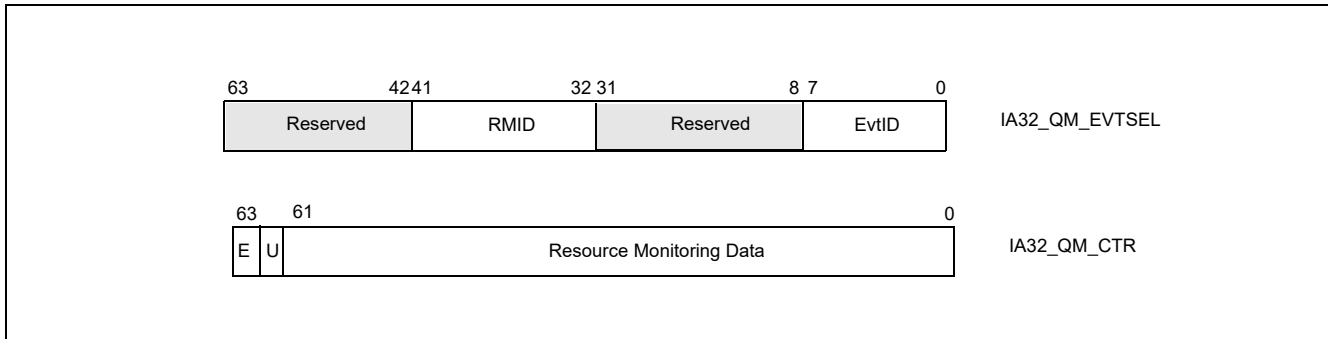


Figure 17-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs

17.18.8 Monitoring Programming Considerations

Figure 17-23 illustrates how system software can program IA32_QOSEVTSEL and IA32_QM_CTR to perform resource monitoring.

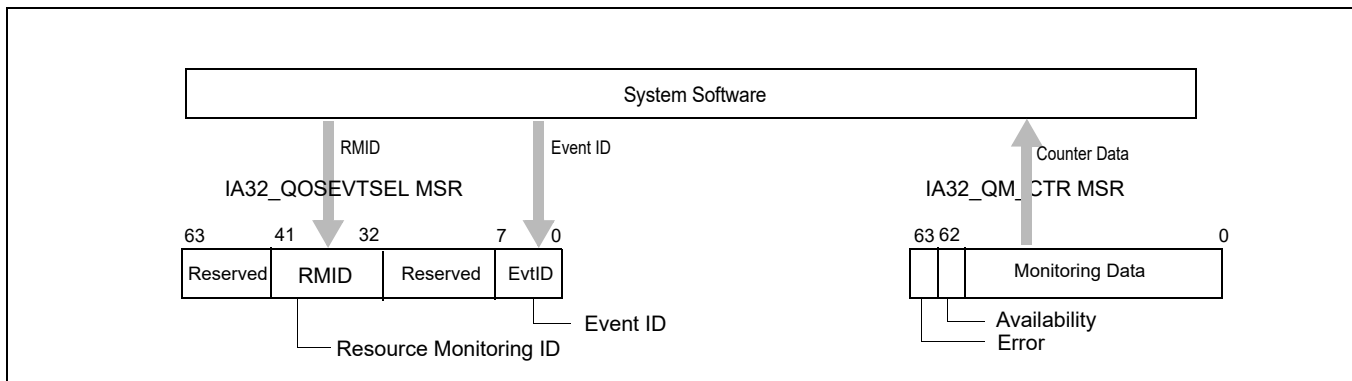


Figure 17-25. Software Usage of Cache Monitoring Resources

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.

17.18.8.1 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.

17.18.8.2 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.

17.18.8.3 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.

17.18.8.4 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.

17.19 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 a subset of communication-focused processors in the Intel Xeon E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Certain Intel Atom processors 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. The CAT and CDP capabilities, where architecturally supported, may be detected and enumerated in software using the *CPUID* instruction, as described in this chapter.

The Intel Xeon Processor Scalable Family introduces 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.

17.19.1 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 given 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.

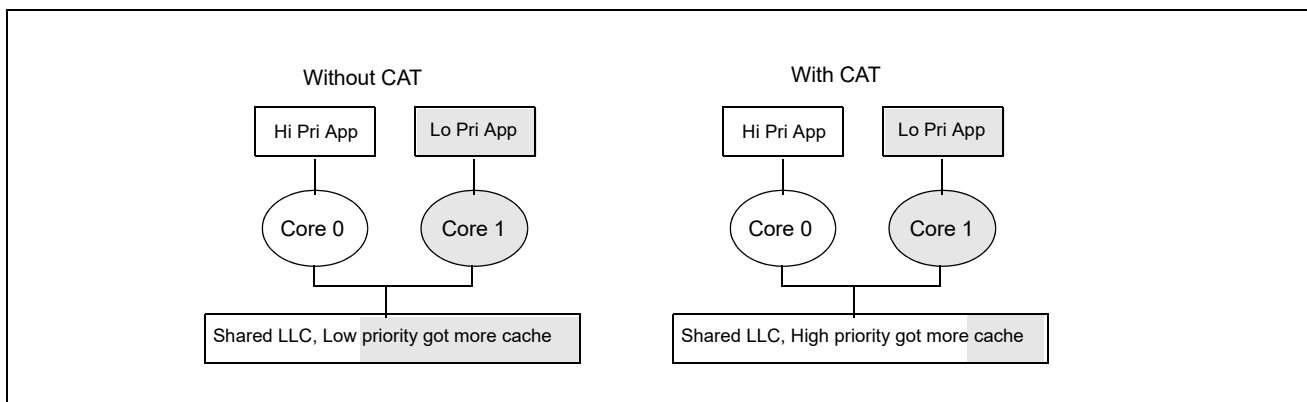


Figure 17-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications

17.19.2 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 resource 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/container/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 bit length 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.

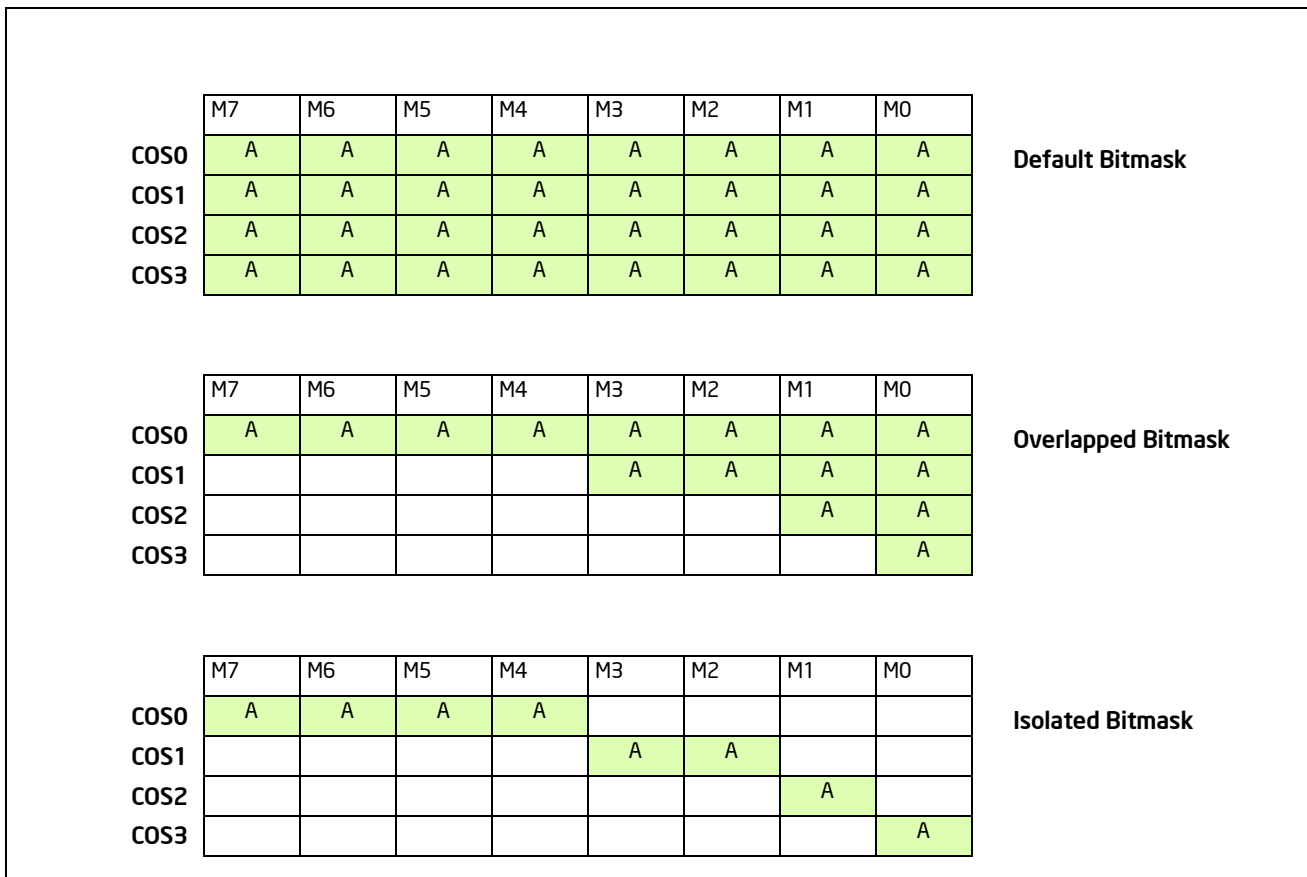


Figure 17-27. Examples of Cache Capacity Bitmasks

Sample cache capacity bitmasks for a bit length 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.

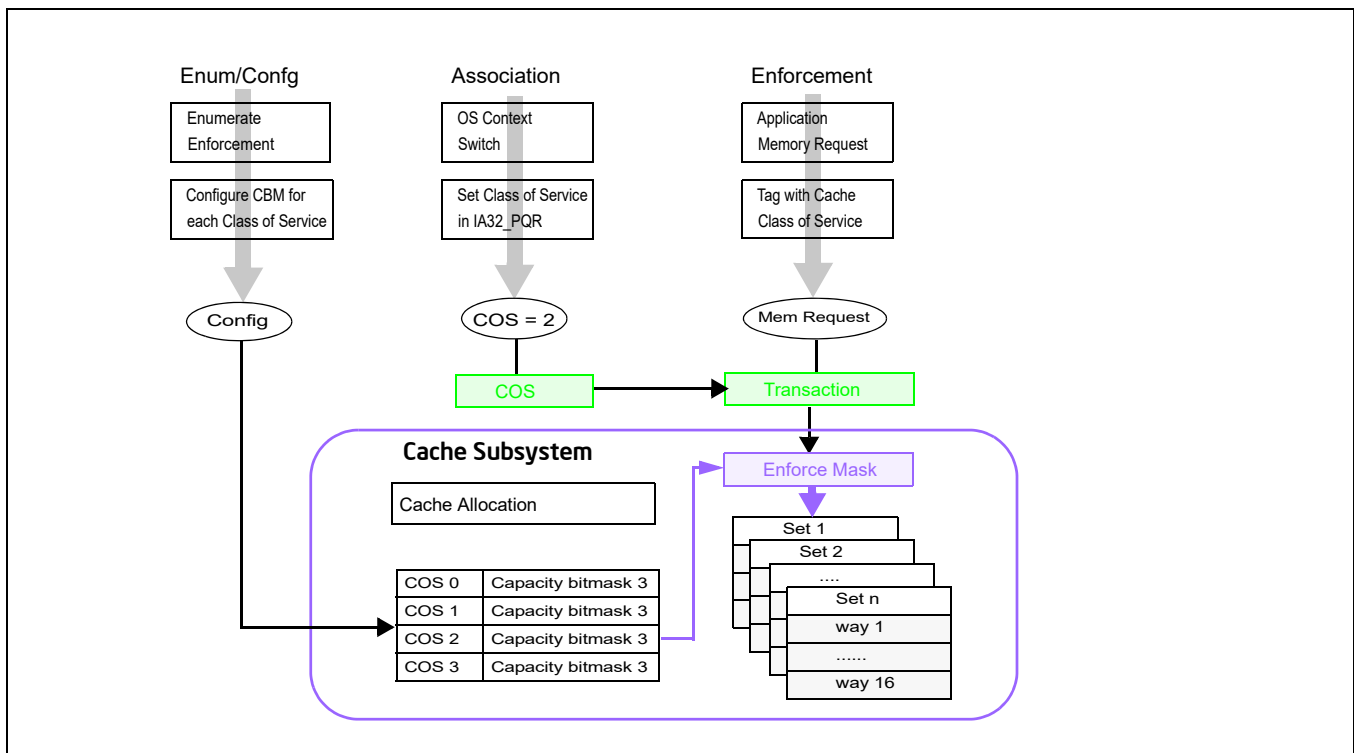


Figure 17-28. Class of Service and Cache Capacity Bitmasks

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 a 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, L2 CAT, and L2 CDP. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to dynamically detect the set of supported features.

17.19.3 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 L2 or L3 cache in a software configurable manner, depending on hardware support, 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). Support for the L2 CDP feature and the L3 CDP features are separately enumerated (via CPUID) and separately controlled (via remapping the L2 CAT MSRs or L3 CAT MSRs respectively). Section 17.19.6.3 and Section 17.19.7 provide details on enumerating, controlling and enabling L3 and L2 CDP respectively, while this section provides a general overview.

The L3 CDP feature was first introduced on the Intel Xeon E5 v4 family of server processors, as an extension to L3 CAT. The L2 CDP feature is first introduced on future Intel Atom family processors, as an extension to L2 CAT.

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 L2 or 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 17.19.5).

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.

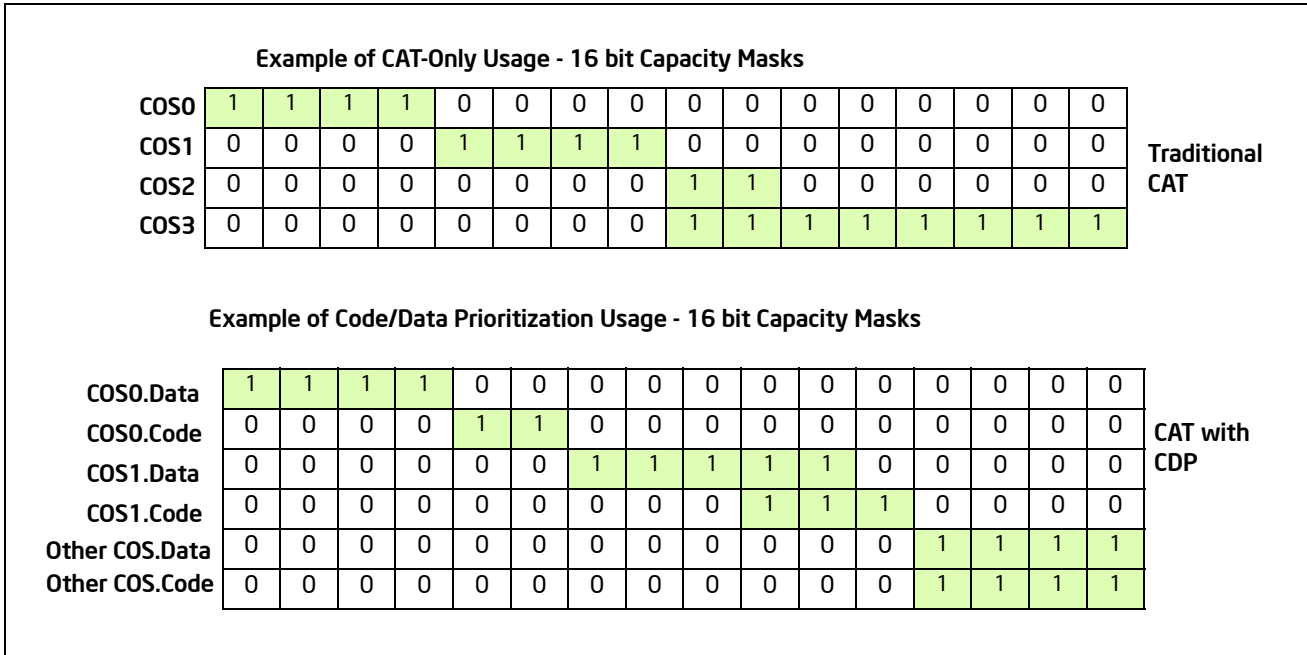


Figure 17-29. Code and Data Capacity Bitmasks of CDP

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) or undefined behavior may result.

17.19.4 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.

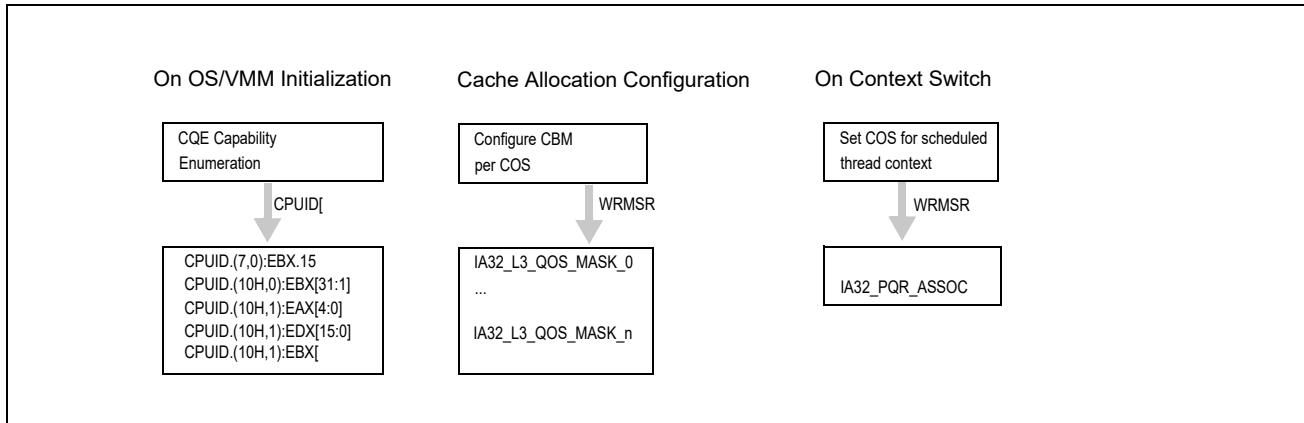


Figure 17-30. Cache Allocation Technology Usage Flow

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.

17.19.4.1 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 17.19.4.2).
- 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 17.19.4.3 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 17.19.4.3 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 CMTI 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 17.19.4.4 for details.

17.19.4.2 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).

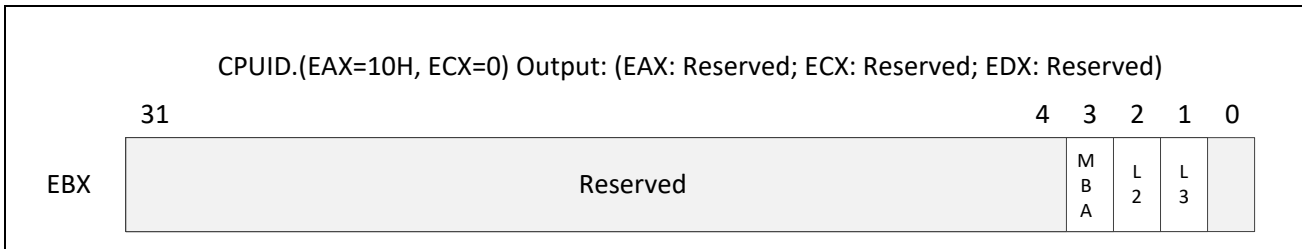


Figure 17-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

- 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:

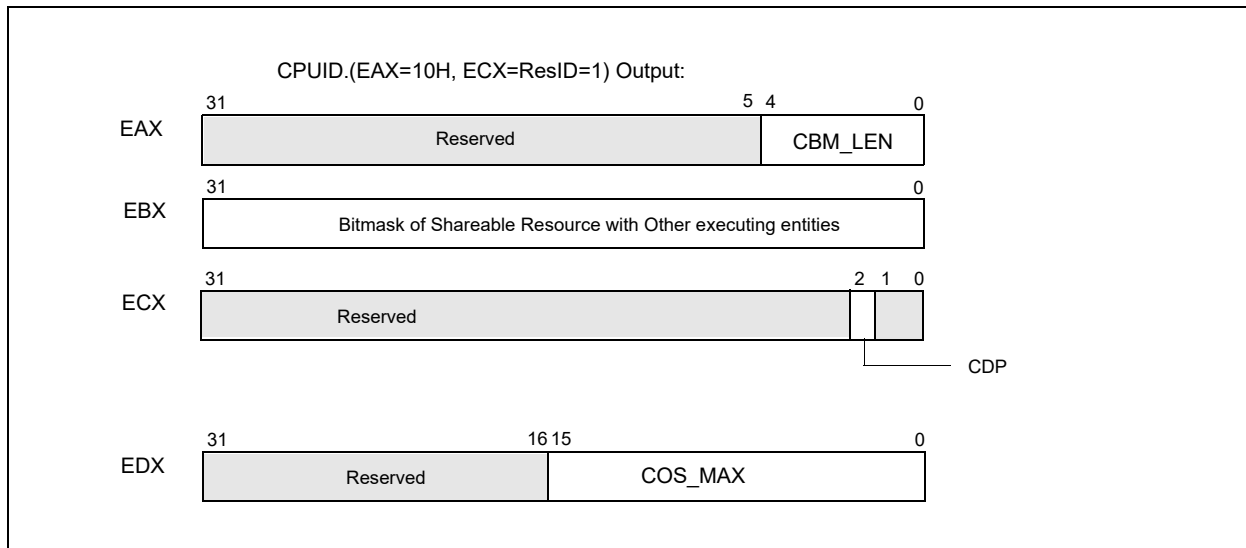


Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration

- 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 L3 Code and Data Prioritization Technology is supported (see Section 17.19.5). 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:

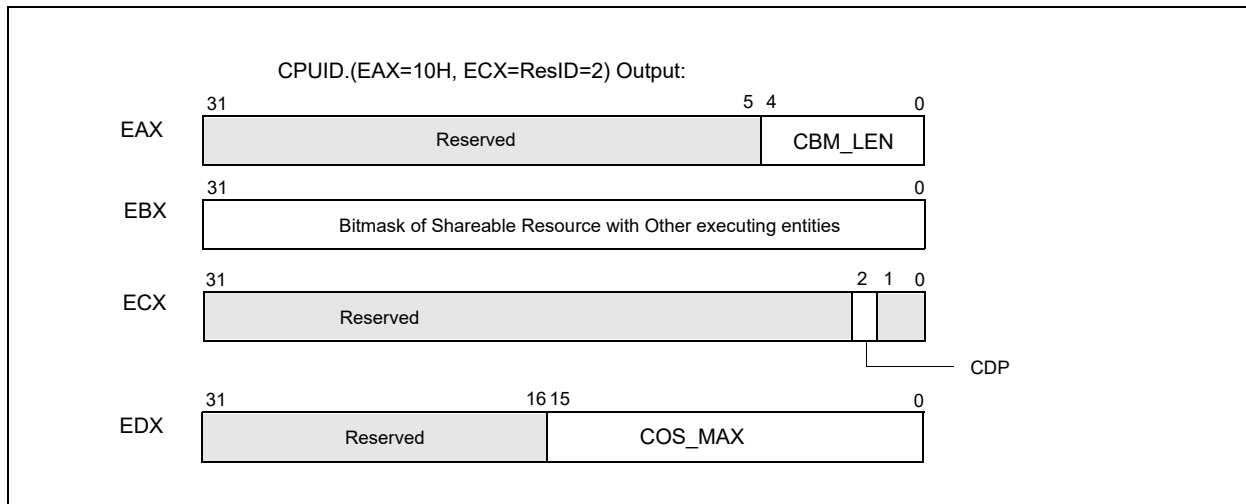


Figure 17-33. L2 Cache Allocation Technology

- 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.CDP[bit 2]: If 1, indicates L2 Code and Data Prioritization Technology is supported (see Section 17.19.6). Other bits of CPUID.(EAX=10H, ECX=2):ECX are 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.

17.19.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of COS supported using CPUID (see Section 17.19.4.2), 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 MSR 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.

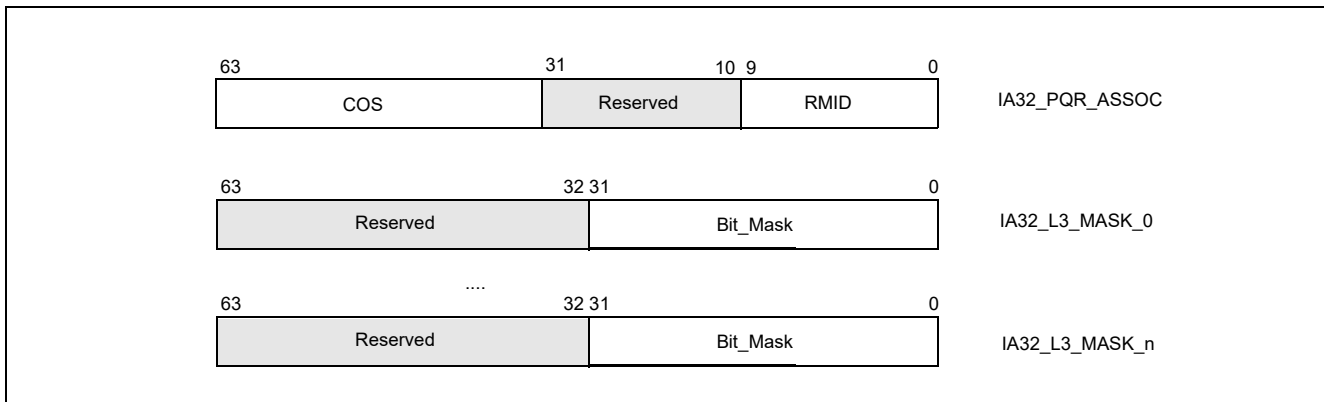


Figure 17-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs

- 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.

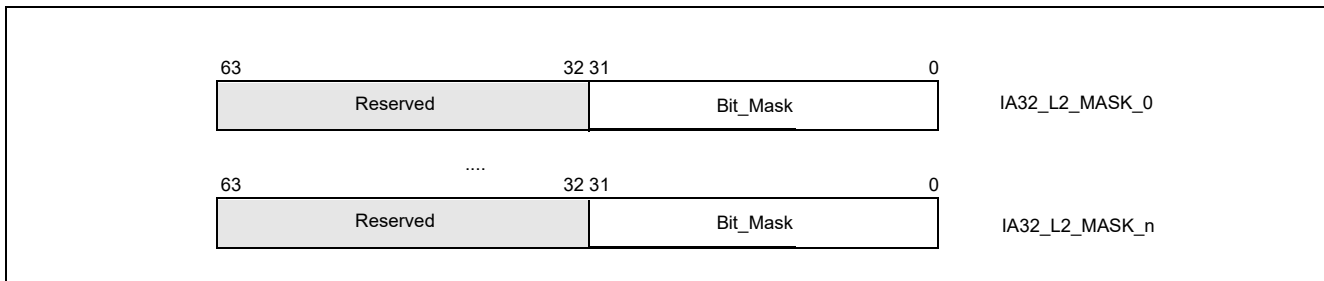


Figure 17-35. IA32_L2_MASK_n MSRs

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".

17.19.4.4 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. In all cases, code and data masks for L2 and L3 CDP should be programmed with at least one bit set.

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 17.19.7, "Introduction to Memory Bandwidth Allocation" for important COS programming considerations including maximum values when using CAT and CDP.

17.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

L3 CDP is an extension of L3 CAT. The presence of the L3 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).

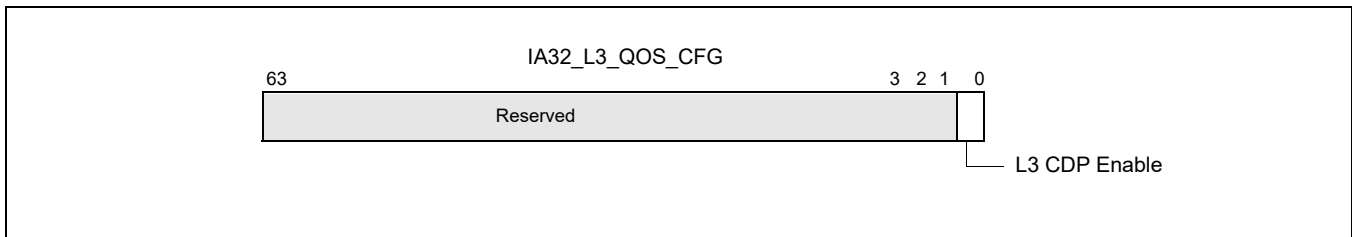


Figure 17-36. Layout of IA32_L3_QOS_CFG

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. The scope of the IA32_L3_QOS_CFG MSR is defined to be the same scope as the L3 cache (e.g., typically per processor socket). Refer to Section 17.19.7 for software considerations while enabling or disabling L3 CDP.

17.19.5.1 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 code 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)`.

17.19.6 Code and Data Prioritization (CDP): Enumerating and Enabling L2 CDP Technology

L2 CDP is an extension of the L2 CAT feature. The presence of the L2 CDP feature is enumerated via CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] (see Figure 17-33). Most of the CPUID.(EAX=10H, ECX=2) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=2):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=2):EDX.COS_MAX_CAT >>1)`.

If CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] =1, the processor supports L2 CDP and provides a new MSR `IA32_L2_QOS_CFG` at address `0C82H`. The layout of `IA32_L2_QOS_CFG` is shown in Figure 17-37. The bit field definition of `IA32_L2_QOS_CFG` are:

- Bit 0: L2 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)`.

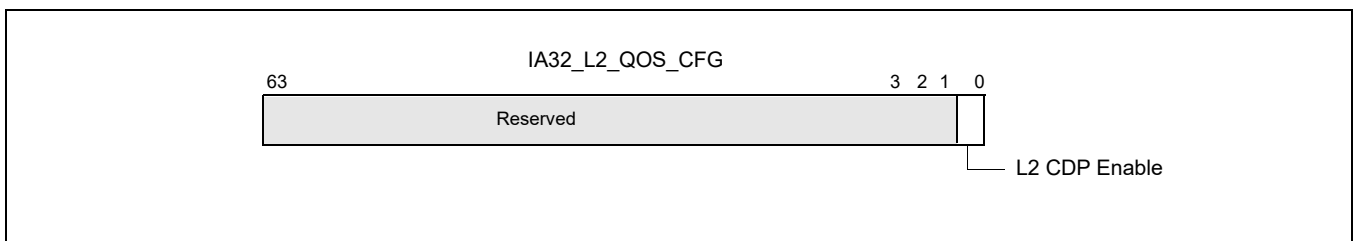


Figure 17-37. Layout of IA32_L2_QOS_CFG

IA32_L2_QOS_CFG default values are all 0s at RESET, and the mask MSRs are all 1s. Hence all logical processors are initialized in COS0 allocated with the entire L2 available and with CDP disabled, until software programs CAT and CDP. The IA32_L2_QOS_CFG MSR is defined at the same scope as the L2 cache, typically at the module level for Intel Atom processors for instance. In processors with multiple modules present it is recommended to program the IA32_L2_QOS_CFG MSR consistently across all modules for simplicity.

17.19.6.1 Mapping Between L2 CDP Masks and L2 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. This remapping is the same as the remapping shown in Table 17-19 for L3 CDP, but for the L2 MSR block (IA32_L2_QOS_MASK_n) instead of the L3 MSR block (IA32_L3_QOS_MASK_n). The same code / data mask mapping algorithm applies to remapping the MSR block between code and data masks.

As with L3 CDP, when L2 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 code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 17-29).

Mask MSR field definitions for L2 CDP remain the same as for L2 CAT. 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).

17.19.6.2 Common L2 and L3 CDP Programming Considerations

Before enabling or disabling L2 or L3 CDP, software should write all 1's to all of the corresponding CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs for the L3 CAT feature). 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.

17.19.6.3 Cache Allocation Technology Dynamic Configuration

All Resource Director Technology (RDT) interfaces including the IA32_PQR_ASSOC MSR, CAT/CDP masks, MBA delay values and CQM/MBM 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.

17.19.6.4 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.

17.19.6.5 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.

17.19.6.6 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).

17.19.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core, and was introduced on the Intel Xeon Processor Scalable Family. 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 17-38.

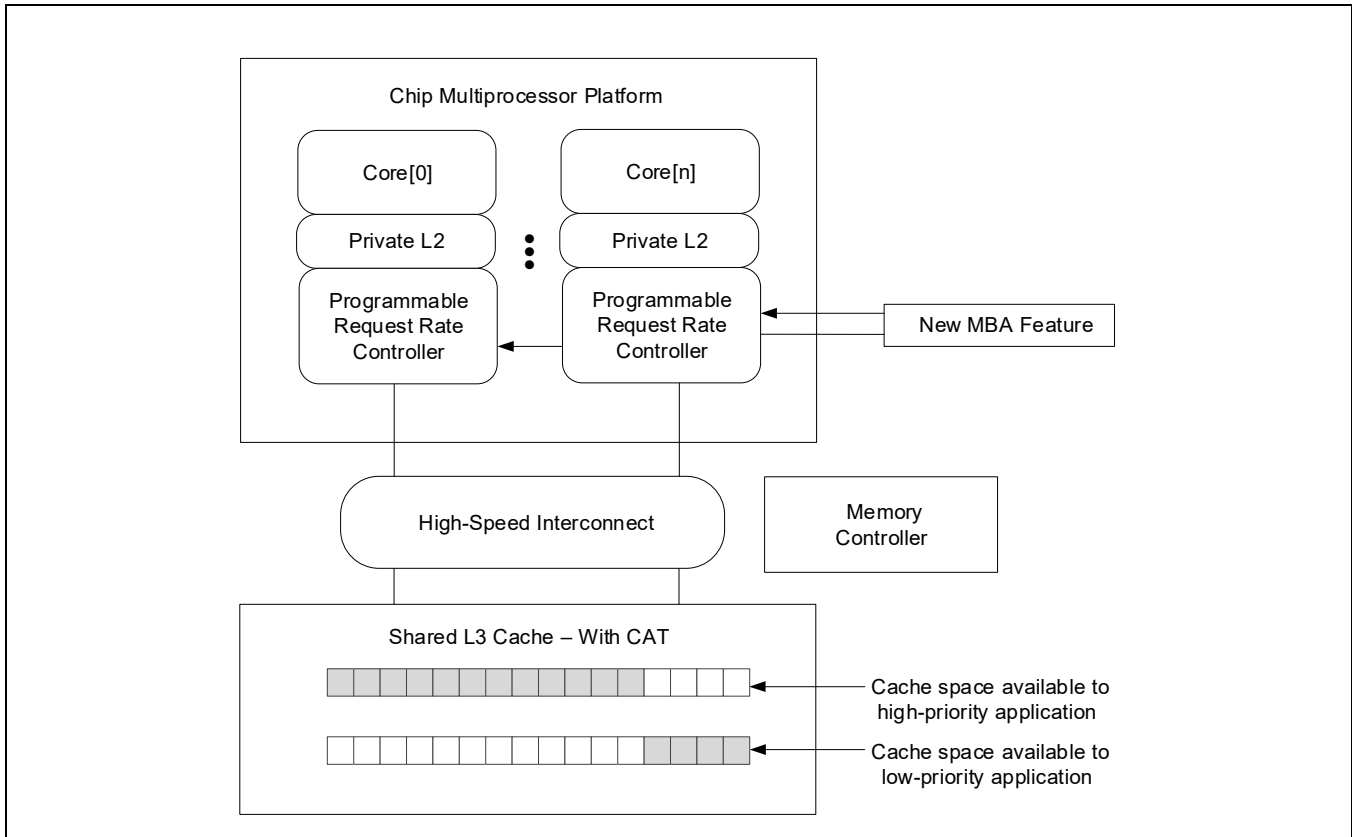


Figure 17-38. A High-Level Overview of the MBA Feature

As shown in Figure 17-38, 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.

17.19.7.1 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 17.19.7.2 on MBA configuration.

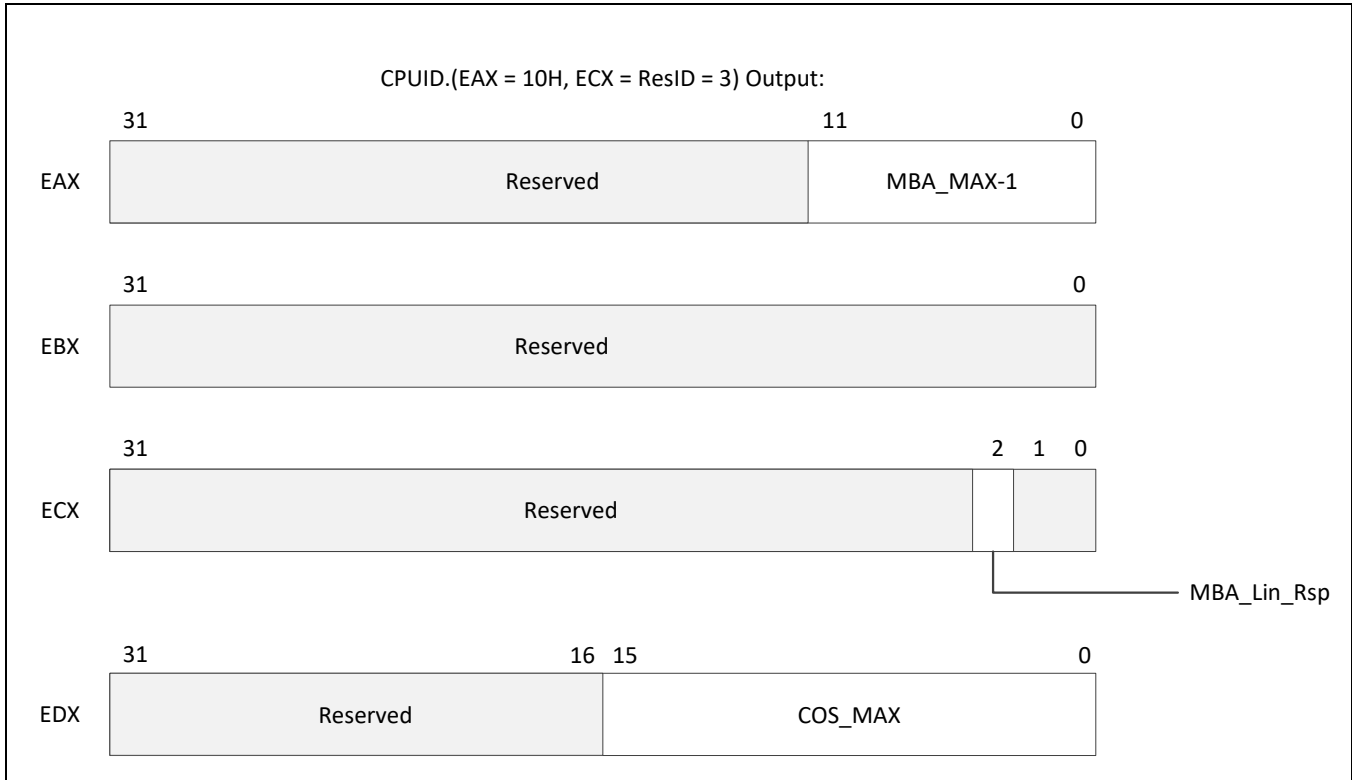


Figure 17-39. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification

17.19.7.2 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 17.19.7.1 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 17.19.7.3 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 17.19.7.1. The throttling values are approximate and do not sum to 100% across CLOS, rather they should be viewed as a maximum bandwidth "cap" per-CLOS.

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 17.19.7.1. 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.

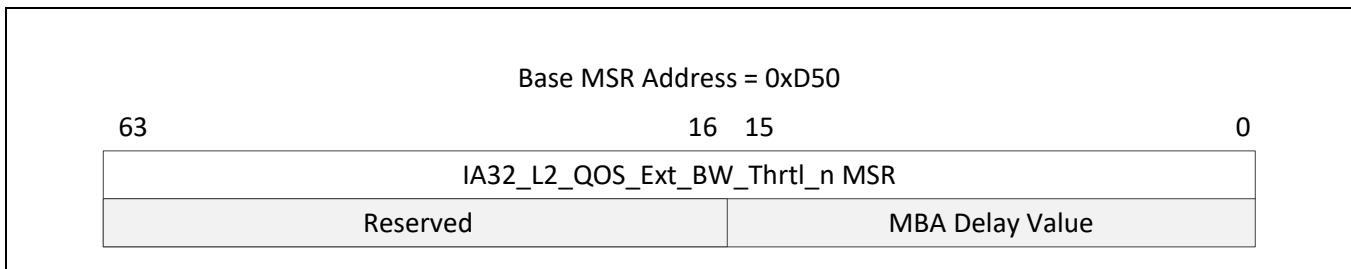


Figure 17-40. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition

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.

17.19.7.3 Memory Bandwidth Allocation Usage Considerations

As the memory bandwidth control that MBA provides is indirect and approximate, 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 band-width 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.

25. 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: Typo corrections as needed. Update to Section 18.3.9.3 "Performance Metrics".

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 18.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)." 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™ Processors and Intel® Xeon® Processors)"
 - Section 18.3.1, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem"
 - Section 18.3.2, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"
 - Section 18.3.3, "Intel® Xeon® Processor E7 Family Performance Monitoring Facility"
 - Section 18.3.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge"
 - Section 18.3.5, "3rd Generation Intel® Core™ Processor Performance Monitoring Facility"
 - Section 18.3.6, "4th Generation Intel® Core™ Processor Performance Monitoring Facility"
 - Section 18.3.7, "5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility"

- Section 18.3.8, “6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.3.9, “Next Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.4, “Performance monitoring (Intel® Xeon™ Phi Processors)”
 - Section 18.4.1, “Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring”
- Section 18.5, “Performance Monitoring (Intel Atom® Processors)”
 - Section 18.5.1, “Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)”
 - Section 18.5.2, “Performance Monitoring for Silvermont Microarchitecture”
 - Section 18.5.3, “Performance Monitoring for Goldmont Microarchitecture”
 - Section 18.5.4, “Performance Monitoring for Goldmont Plus Microarchitecture”
 - Section 18.5.5, “Performance Monitoring for Tremont Microarchitecture”
- Section 18.6, “Performance Monitoring (Legacy Intel Processors)”
 - Section 18.6.1, “Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
 - Section 18.6.2, “Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)”
 - Section 18.6.3, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)”
 - Section 18.6.4, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
 - Section 18.6.4.5, “Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
 - Section 18.6.5, “Performance Monitoring and Dual-Core Technology”
 - Section 18.6.6, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache”
 - Section 18.6.7, “Performance Monitoring on L3 and Caching Bus Controller Sub-Systems”
 - Section 18.6.8, “Performance Monitoring (P6 Family Processor)”
 - Section 18.6.9, “Performance Monitoring (Pentium Processors)”
- Section 18.7, “Counting Clocks”
- Section 18.8, “IA32_PERF_CAPABILITIES MSR Enumeration”
- Section 18.9, “PEBS Facility”

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, Kaby Lake and Coffee Lake microarchitectures support version ID 4.

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support version ID 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 to software 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]. Note that this may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
- 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. Note the number of IA32_PERFEVTSELx MSRs may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
- 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.

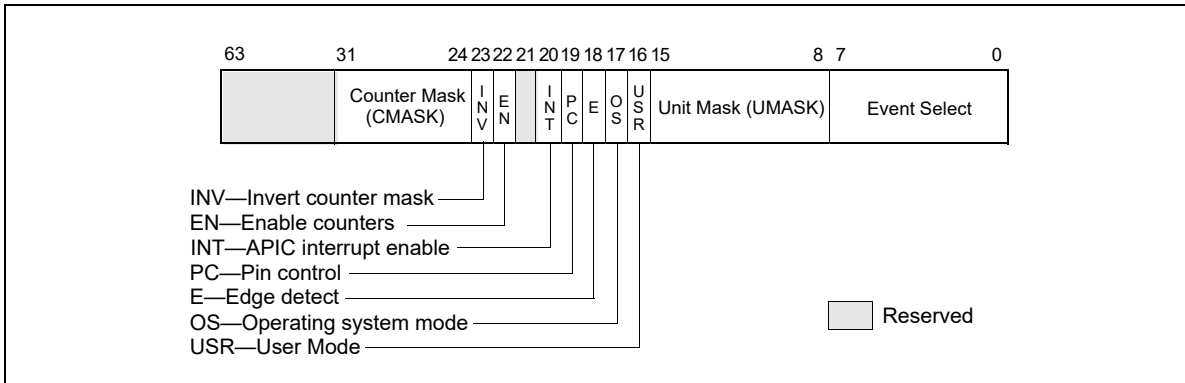


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

- **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)** — Beginning with Sandy Bridge microarchitecture, this bit is reserved (not writeable). On processors based on previous microarchitectures, the logical processor toggles the PMi pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PMi 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
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H
7	Topdown Slots	01H	A4H

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The number of architectural events is reported through CPUID.0AH:EAX[31:24], while non-zero bits in CPUID.0AH:EBX indicate any architectural 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.

- **Topdown Slots** — Event select A4H, Umask 01H

This event counts the total number of available slots for an unhalted logical processor.

The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core, in processors that support Intel Hyper-Threading Technology.

Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

NOTE

Programming decisions or software precisions 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 MSR:
 - 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:

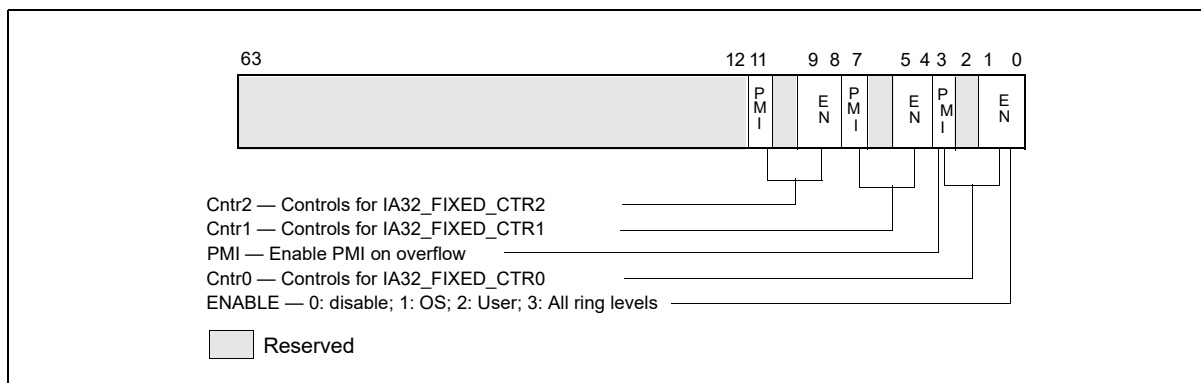


Figure 18-2. Layout of IA32_FIXED_CTR_CTRL MSR

- 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.

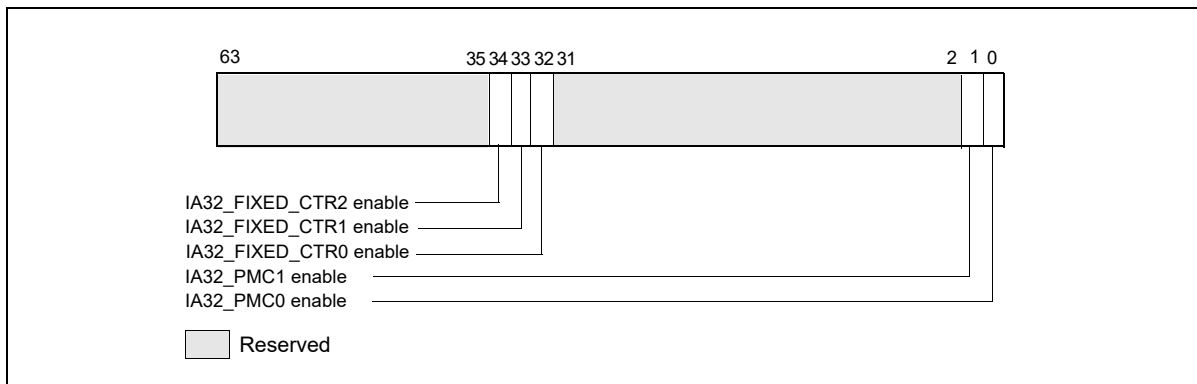


Figure 18-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

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
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.
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.
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_FIXED_CTR3	30CH	TOPDOWN.SLOTS	This event counts the number of available slots for an unhalted logical processor. The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core. Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

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.

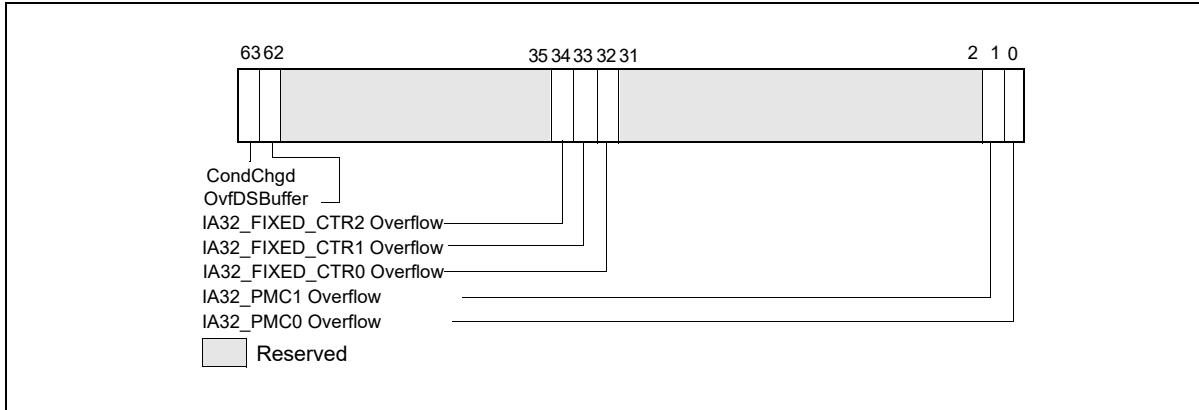


Figure 18-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

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.

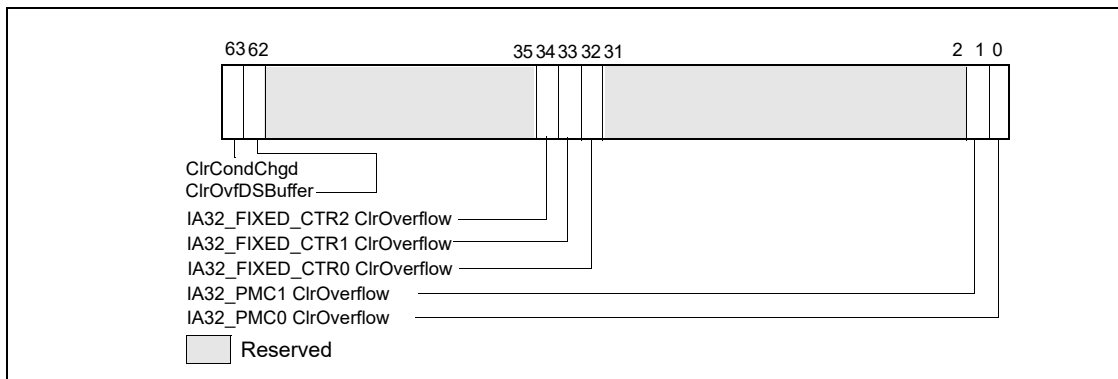


Figure 18-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

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.

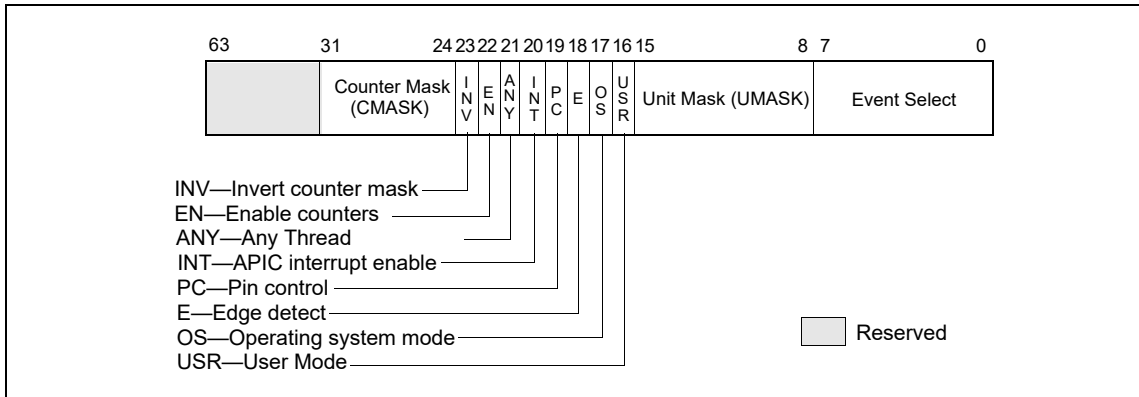


Figure 18-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

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.

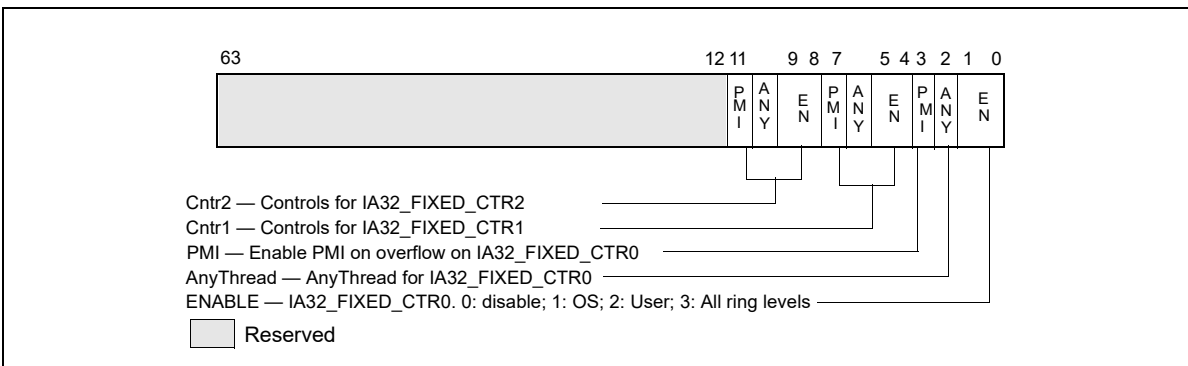


Figure 18-7. IA32_PERF_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides an **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). In addition, the number of counters may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

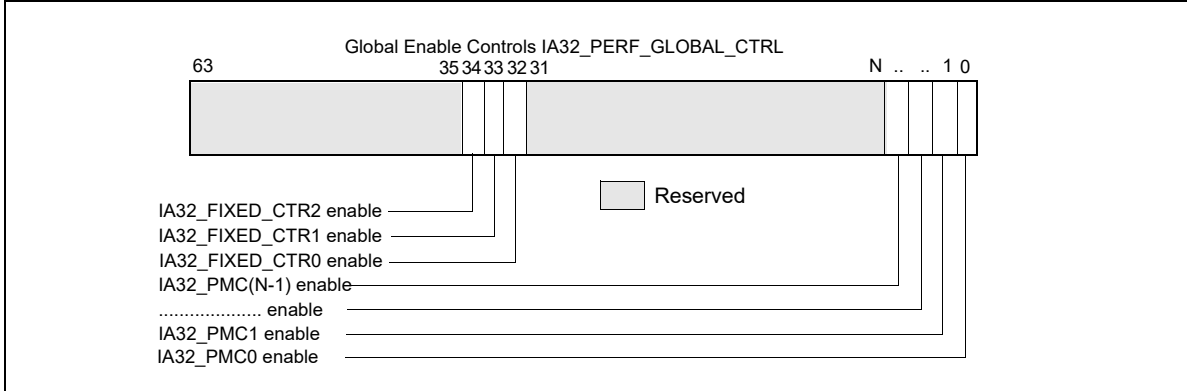


Figure 18-8. Layout of Global Performance Monitoring Control MSR

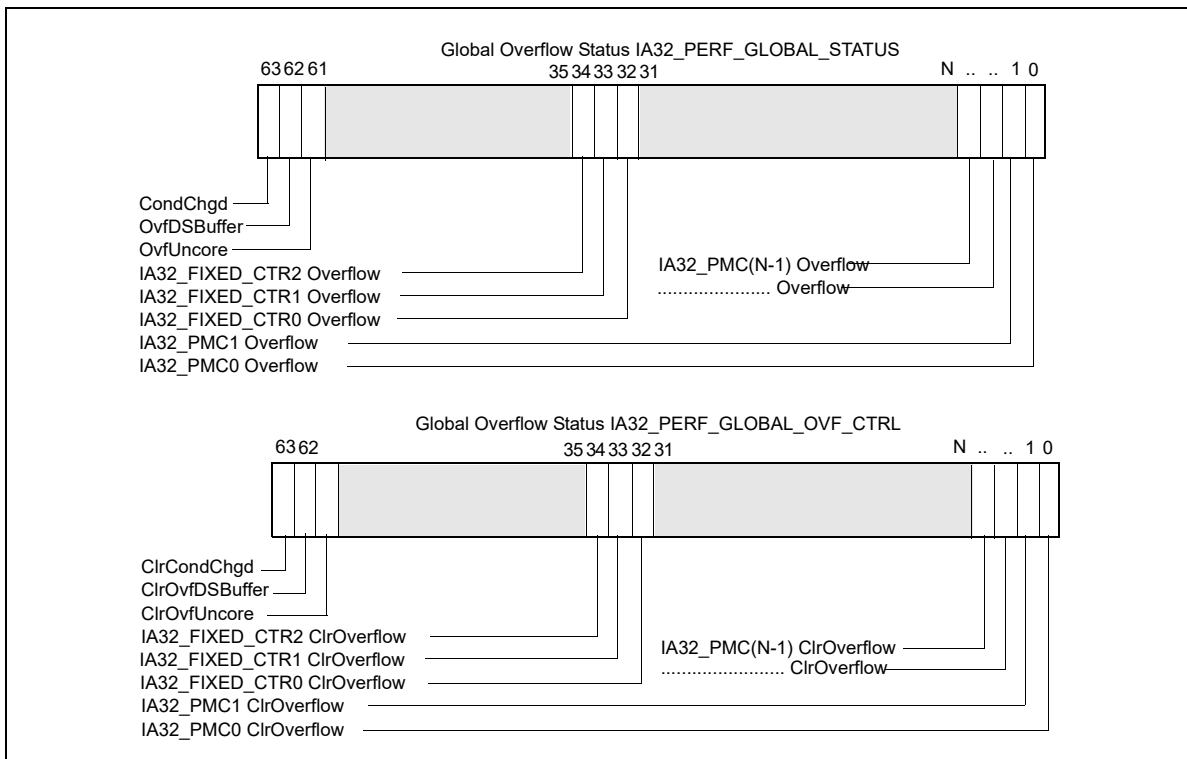


Figure 18-9. Global Performance Monitoring Overflow Status and Control MSRs

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 serves as a 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:

$$\text{— } (\text{IA32_DEBUGCTL.LBR} \ \& \ (\text{!IA32_PERF_GLOBAL_STATUS.LBR_FRZ})) = 1$$

- 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
- $(IA32_PERF_FIXED_CTRL.ENi \ \& \ IA32_PERF_GLOBAL_CTRL.FCi \ \& \ (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1$ 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.Freeze_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).

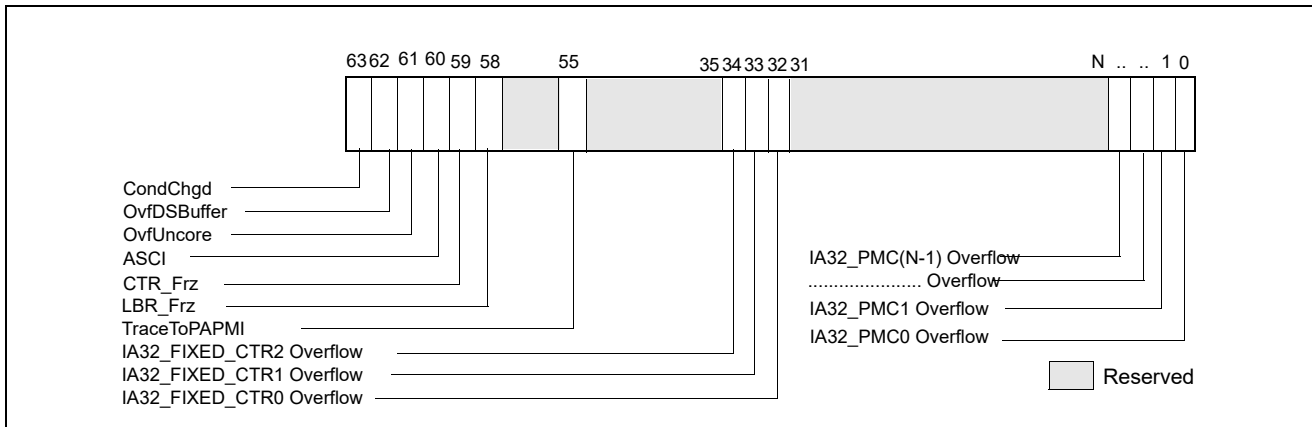


Figure 18-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural Perfmon Version 4

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.

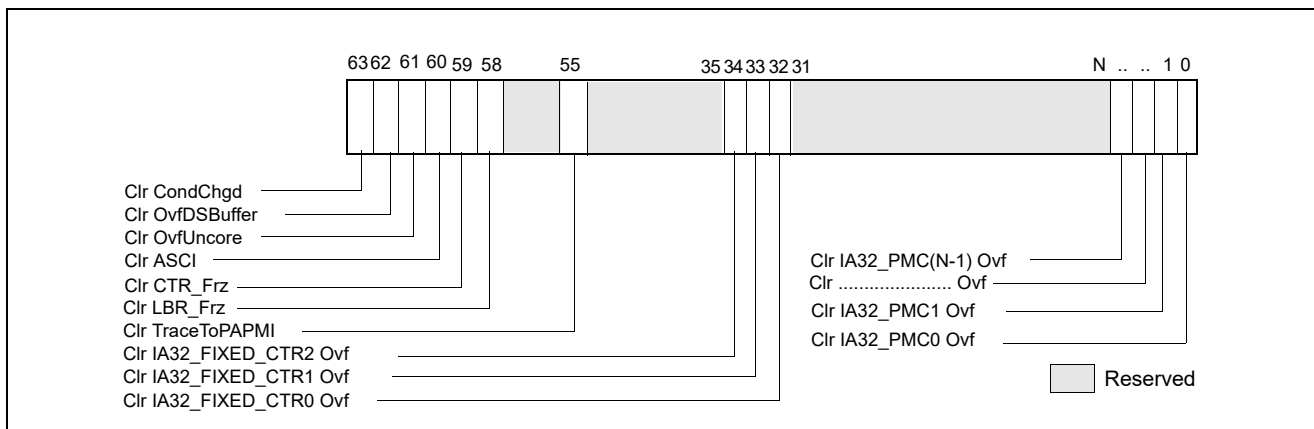


Figure 18-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4

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.

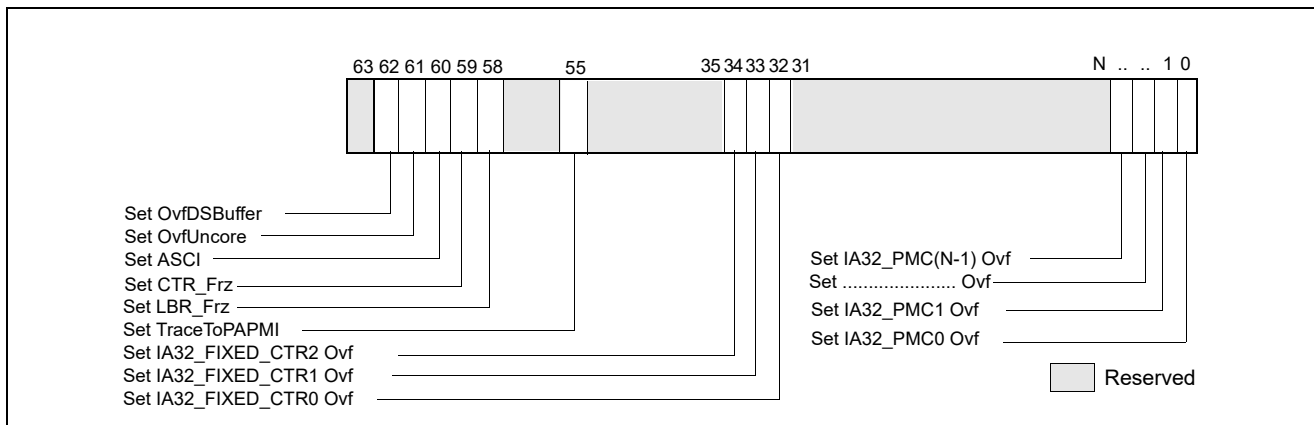


Figure 18-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4

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>

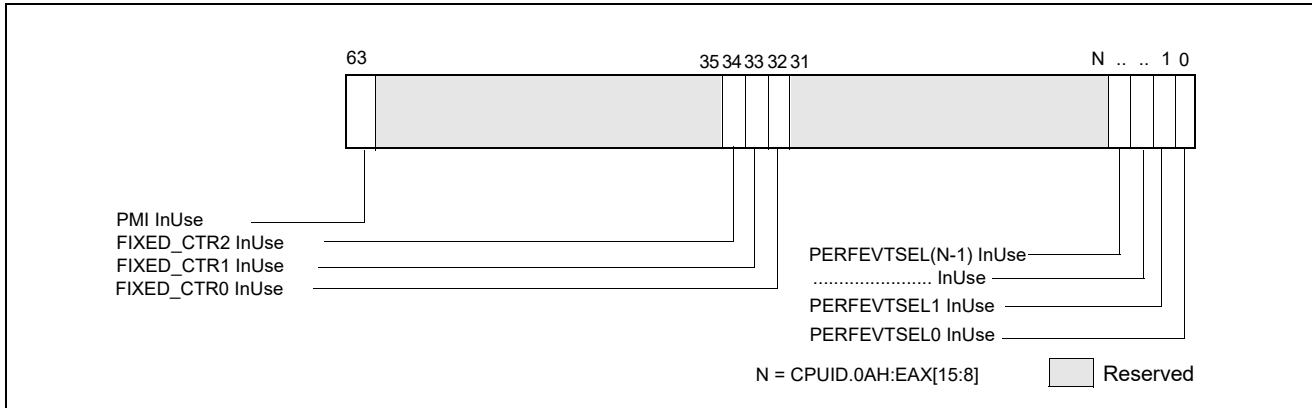


Figure 18-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural Perfmon Version 4

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 63]: 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.
 - Any IA32_PEBS_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

18.2.5 Architectural Performance Monitoring Version 5

Processors supporting architectural performance monitoring version 5 also support versions 1, 2, 3 and 4, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 5 provides the following enhancements:

- Deprecation of Anythread mode, see Section 18.2.5.1.
- Individual enumeration of Fixed counters in CPUID.0AH, see Section 18.2.5.2.

18.2.5.1 AnyThread Mode Deprecation

With Architectural Performance Monitoring Version 5, a processor that supports AnyThread mode deprecation is enumerated by CPUID.0AH.EDX[15]. If set, software will not have to follow guidelines in Section 18.2.3.1.

18.2.5.2 Fixed Counter Enumeration

With Architectural Performance Monitoring Version 5, register CPUID.0AH.ECX indicates Fixed Counter enumeration. It is a bit mask which enumerates the supported Fixed Counters in a processor. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor:

```
FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i);
```

18.2.6 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 18-63.

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™ PROCESSORS AND INTEL® XEON® PROCESSORS)

18.3.1 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem

Intel Core i7 processor family¹ 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 19-31. 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.

1. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem; the performance monitoring facilities described in this section generally also apply.

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 18-15.

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.

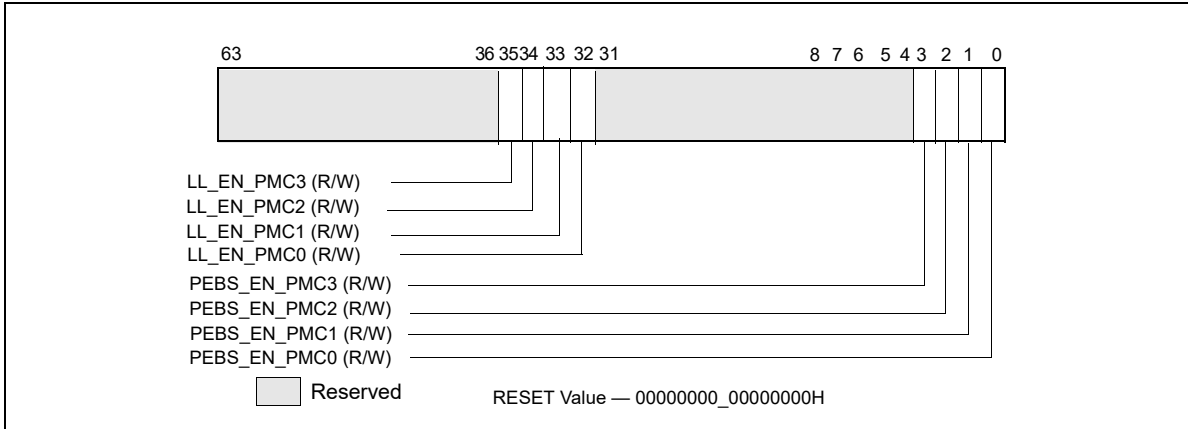


Figure 18-15. Layout of IA32_PEBS_ENABLE MSR

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 18-63).

The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 18-63). 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 18-3, 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 18-3. 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

Table 18-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
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 18-78. 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 18-16.

- **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.

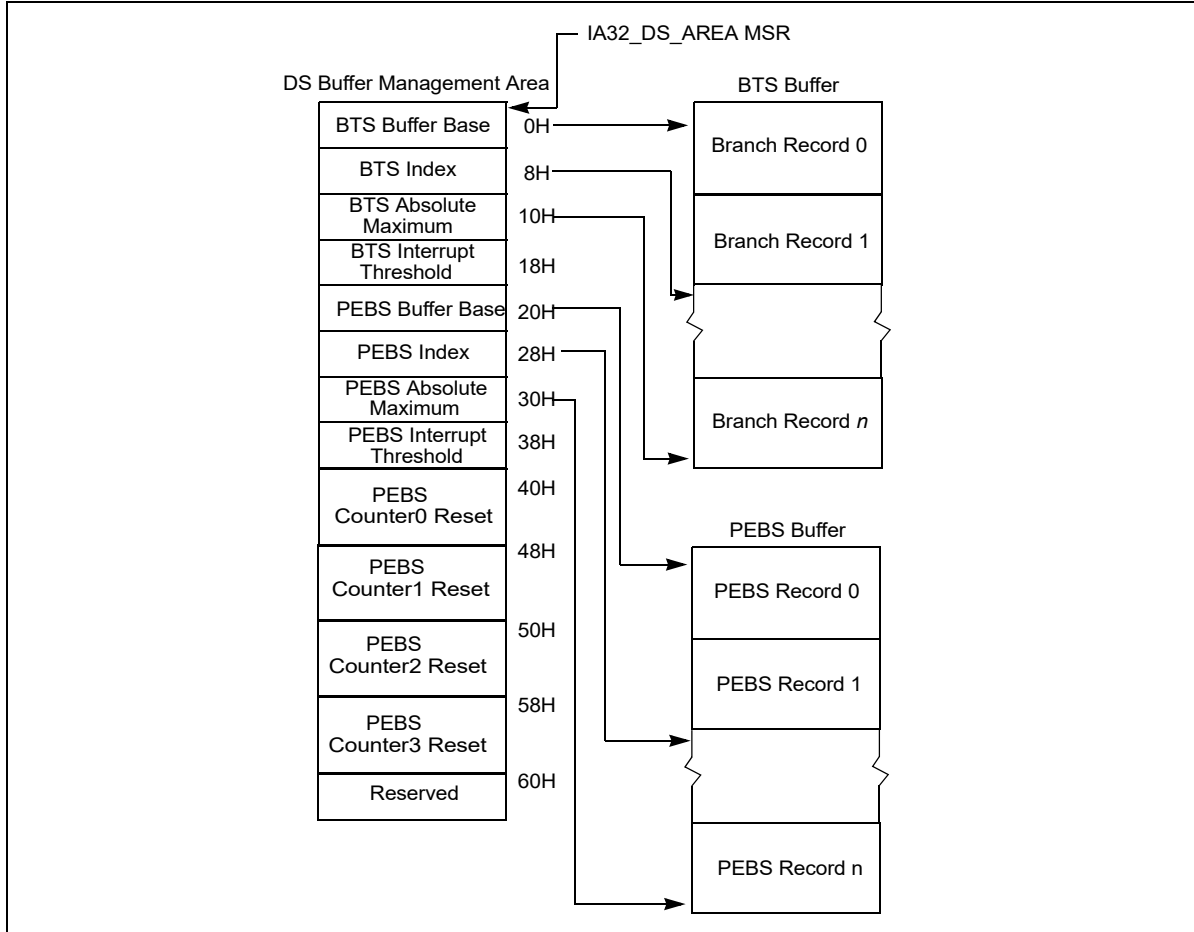


Figure 18-16. PEBS Programming Environment

- 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 18.3.1.2). 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.

18.3.1.1.2 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 18-3. 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 18-3, 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 18-4. 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 18-4. 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 were 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 were serviced by another processor core with a cross core snoop where no modified copies were found.
07H ¹	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and were serviced by another core with a cross core snoop where modified copies were found.
08H	Reserved/L3 MISS. Local homed requests that missed the L3 cache and were serviced by forwarded data following a cross package snoop where no modified copies were found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and were 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:

- Bit 7 is supported only for processors with a 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 18-17.

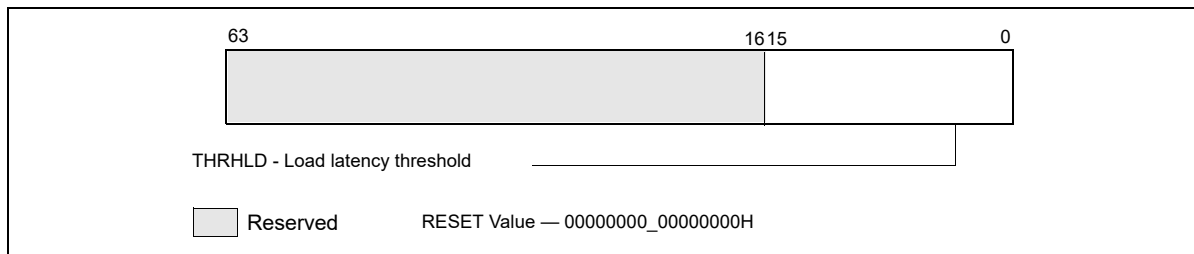


Figure 18-17. Layout of MSR_PEBS_LD_LAT MSR

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.

18.3.1.1.3 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 18-5 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 18-5. 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 18-18. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.

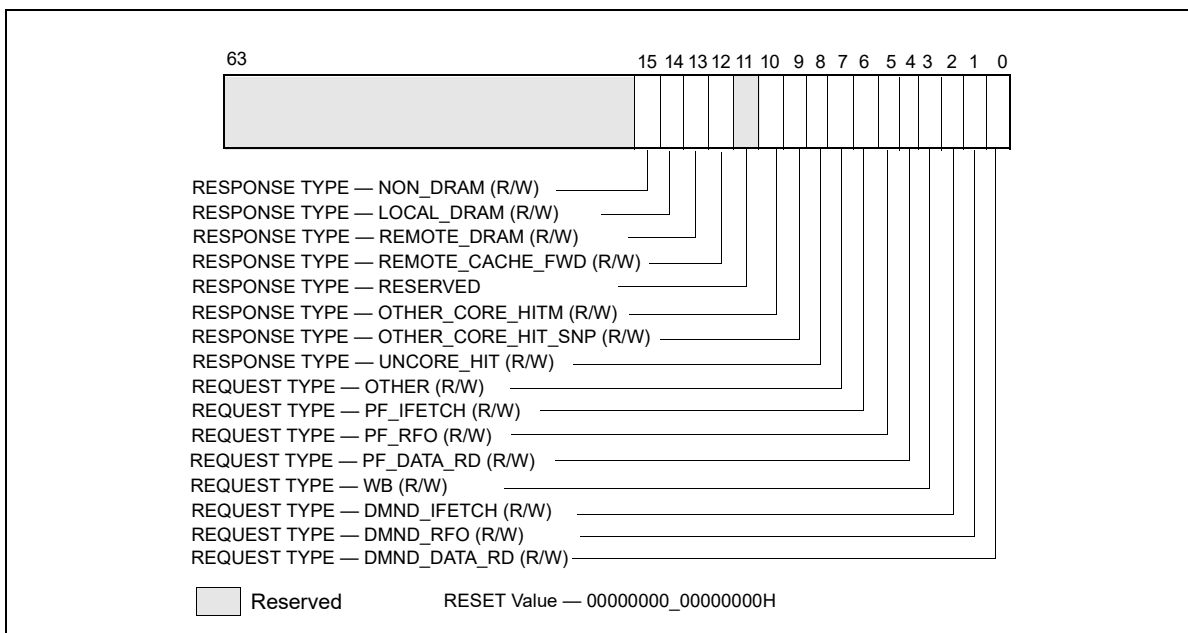


Figure 18-18. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	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	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	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
OTHER	7	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	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	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	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	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	L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	Non-DRAM requests that were serviced by IOH.

18.3.1.2 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.

18.3.1.2.1 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 18-19 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.

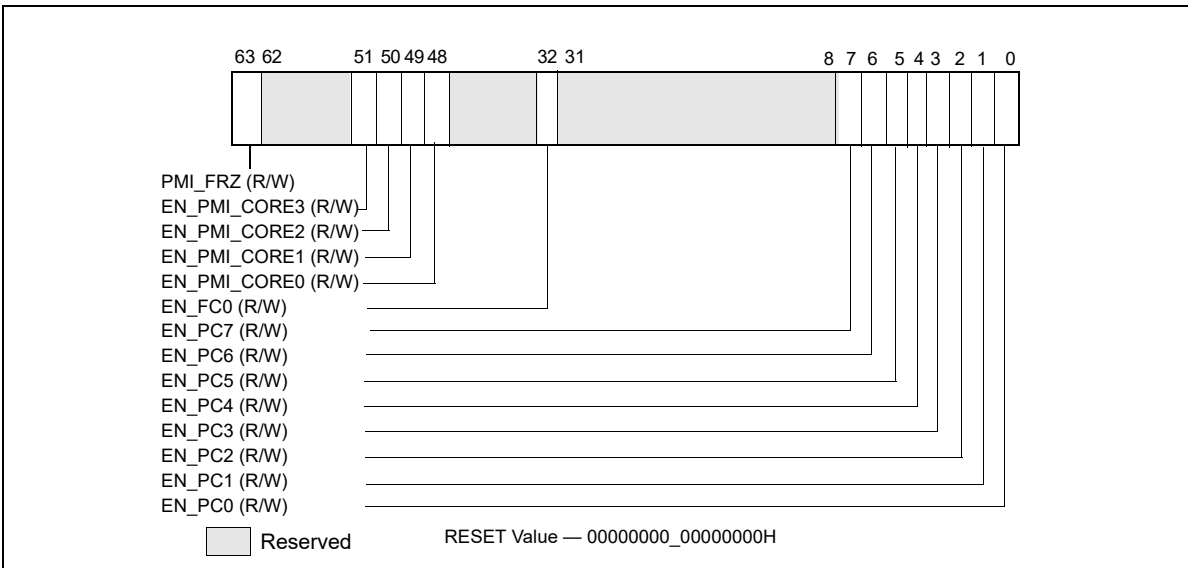


Figure 18-19. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

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 18-20 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.

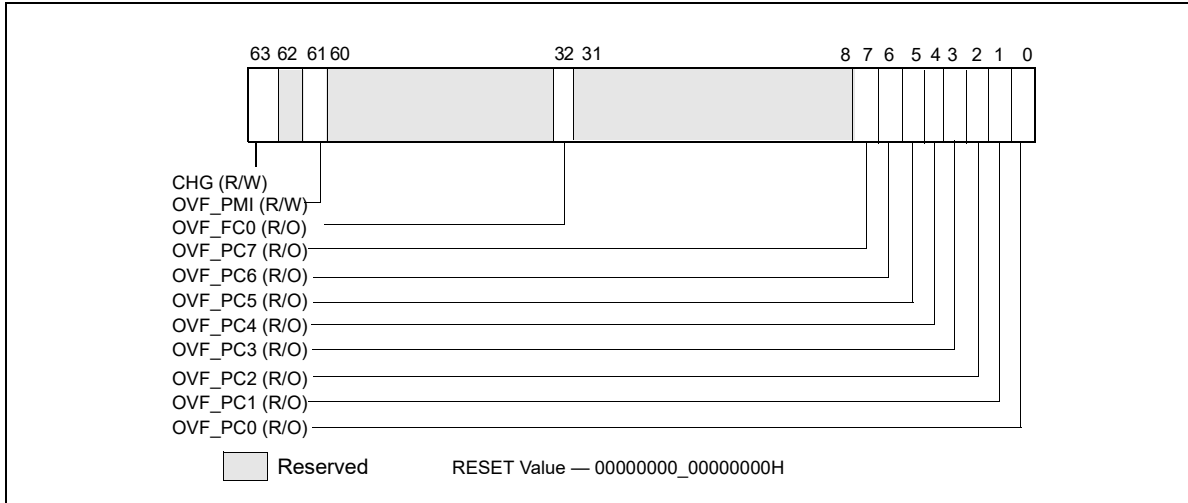


Figure 18-20. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 18-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

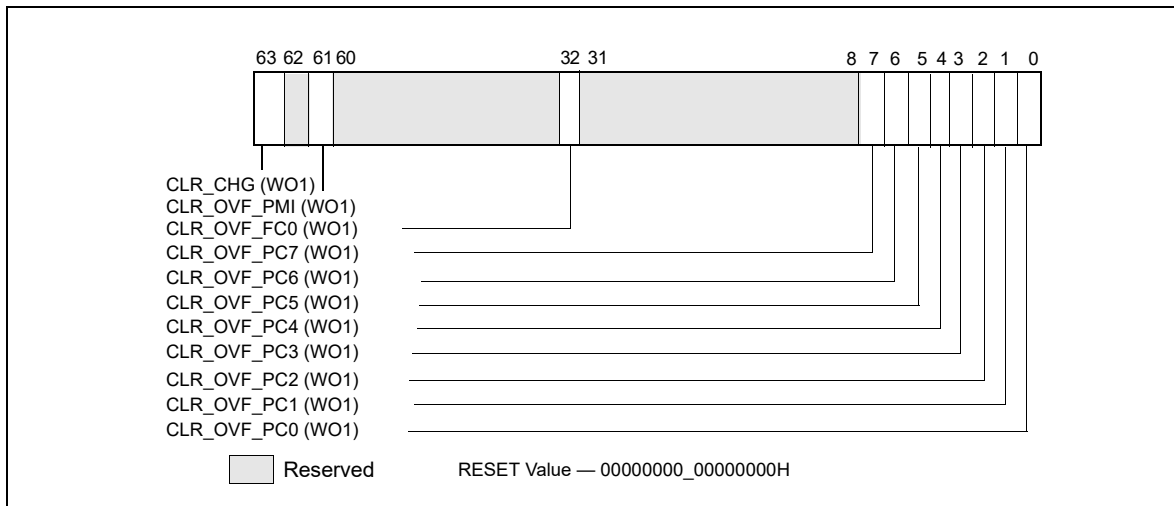


Figure 18-21. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- 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.

18.3.1.2.2 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 18-22 shows the layout of MSR_UNCORE_PERFEVTSELx.

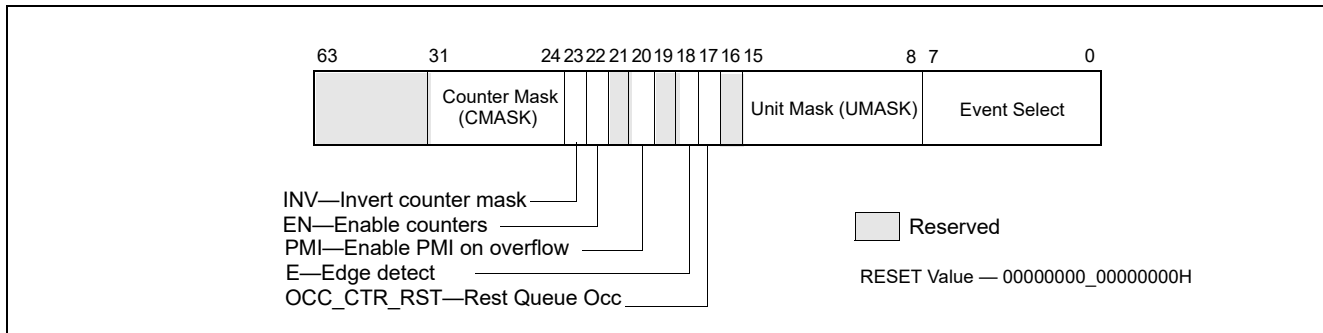


Figure 18-22. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- 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 18-23 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

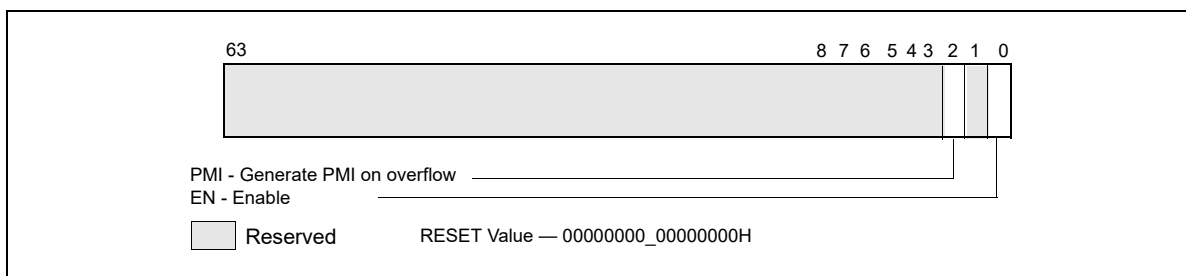


Figure 18-23. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- 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_PerfCnt) and the fixed-function counter (MSR_UNCORE_FixedCnt0) 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.

18.3.1.2.3 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 18-24.

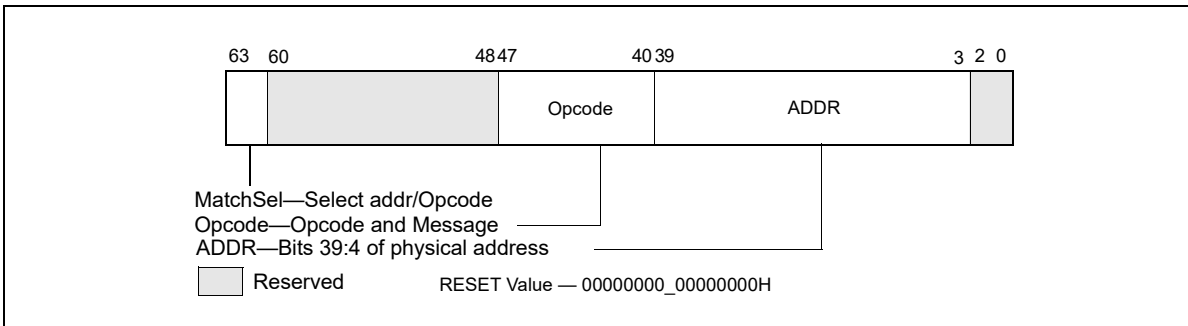


Figure 18-24. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- 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 18-7 lists the encodings supported in the opcode field.

Table 18-7. 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.

18.3.1.3 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 18-25.

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.

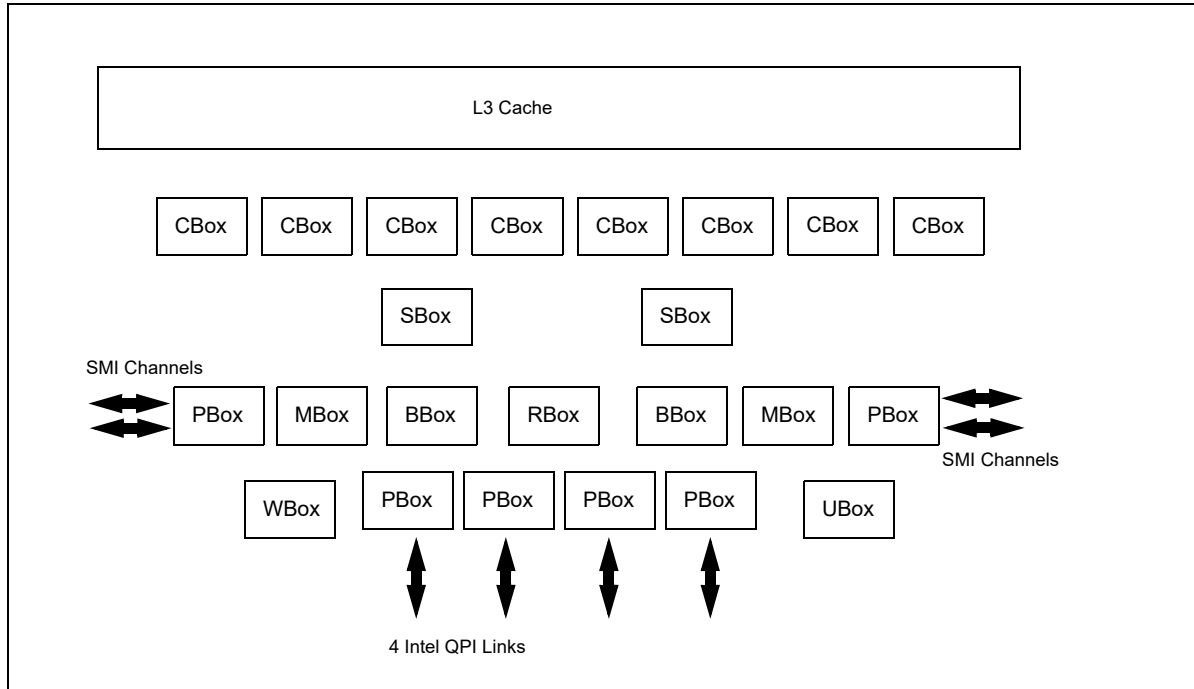


Figure 18-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-8 summarizes the number MSRs for uncore PMU for each box.

Table 18-8. 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 MSR definitions 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 2-17 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 than 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”.

18.3.2 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 18.6.3 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 18-5 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 19-31.

The load latency facility is the same as described in Section 18.3.1.1.2, 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 18-13.

18.3.3 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 series¹. 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 subsystem is similar to that shown in Figure 18-25, with the additional capability that up to 10 C-Box units are supported.

1. Exceptions are indicated for event code 0FH in Table 19-23; and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 18-13.

Table 18-9 summarizes the number MSRs for uncore PMU for each box.

Table 18-9. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

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

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”.

18.3.4 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 18.3.1.1 and Section 18.6.3, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 18-10.

Table 18-10. 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 determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
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 determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM. 	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM. 	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12.	See Table 18-78.	IA32_PMC4-IA32_PMC7 do not support PEBS.

Table 18-10. Core PMU Comparison (Contd.)

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
PEBS-Load Latency	See Section 18.3.4.4.2; <ul style="list-style-type: none"> ▪ Data source encoding ▪ STLB miss encoding ▪ Lock transaction encoding 	Data source encoding	
PEBS-Precise Store	Section 18.3.4.4.3	No	
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.

18.3.4.1 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).

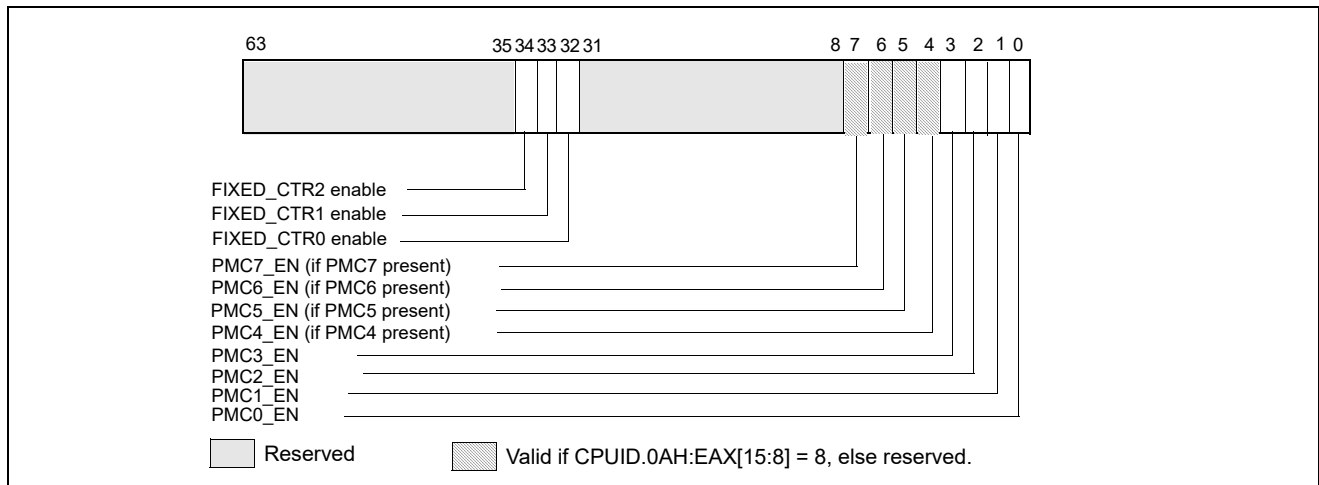


Figure 18-26. IA32_PERF_GLOBAL_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

Figure 18-42 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 18-27). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

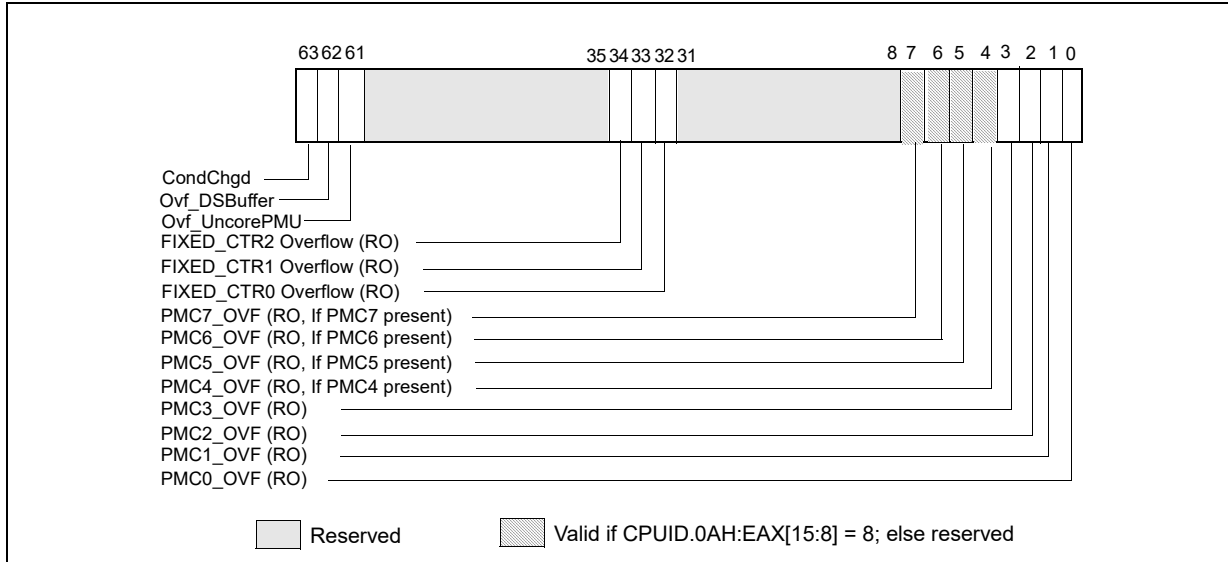


Figure 18-27. IA32_PERF_GLOBAL_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

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 18-28). 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.

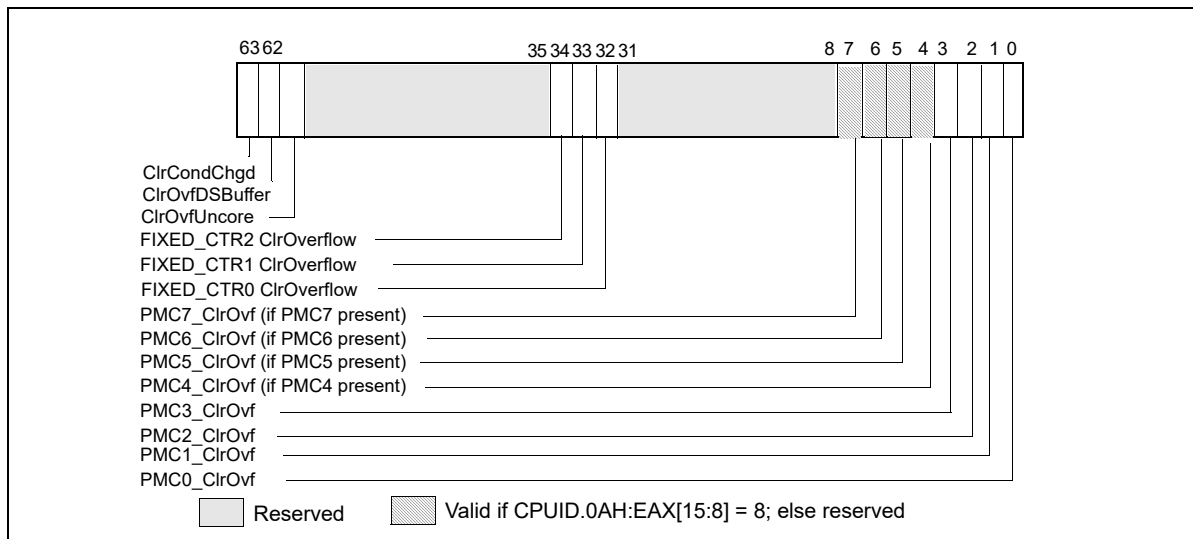


Figure 18-28. IA32_PERF_GLOBAL_OVF_CTRL MSR in Intel microarchitecture code name Sandy Bridge

18.3.4.2 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 the number of counters visible to software.

If a processor core is shared by two logical processors, each logical processors can access up to four 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, up to eight general-purpose counters are visible. If CPUID.0AH:EAX[15:8] reports 8 counters, then IA32_PMC4-IA32_PMC7 would 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.

18.3.4.3 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].

18.3.4.4 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 18-11.

Table 18-11. 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 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-29	Figure 18-15	
PEBS record layout	Physical Layout same as Table 18-3.	Table 18-3	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 18-12.	See Table 18-78.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-4	
PEBS-Precise Store	Yes; see Section 18.3.4.4.3.	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.

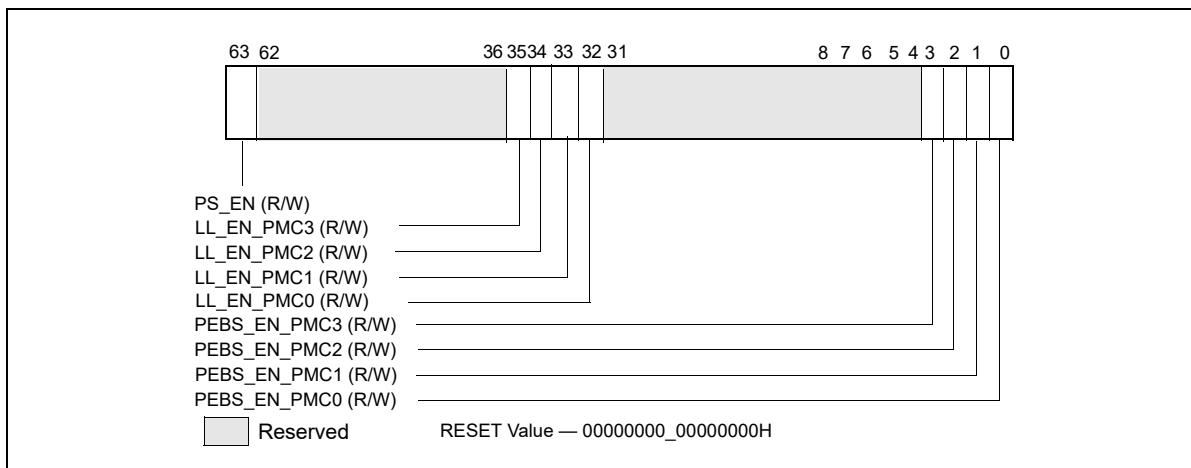


Figure 18-29. Layout of IA32_PEBS_ENABLE MSR

18.3.4.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 18-3, 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 18-4. 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 18-12.

Table 18-12. PEBS Performance Events for Intel® Microarchitecture Code Name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	COH	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	DOH	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.

18.3.4.4.2 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 18-3 and Section 18.3.4.4.1. 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_RETIRE 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 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

Table 18-13. Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 18-4
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 18-17.

18.3.4.4.3 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_PERFVTSEL3. 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 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 18-14. 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

18.3.4.4.4 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. On processors based on Intel microarchitecture code name Sandy Bridge skid is significantly reduced, and can be as little as one instruction. On future implementations PDIR may eliminate skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and processors based on Sandy Bridge microarchitecture 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.

18.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be 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 18-15 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 18-15. Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMCO-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 18-30 and Figure 18-31. 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.

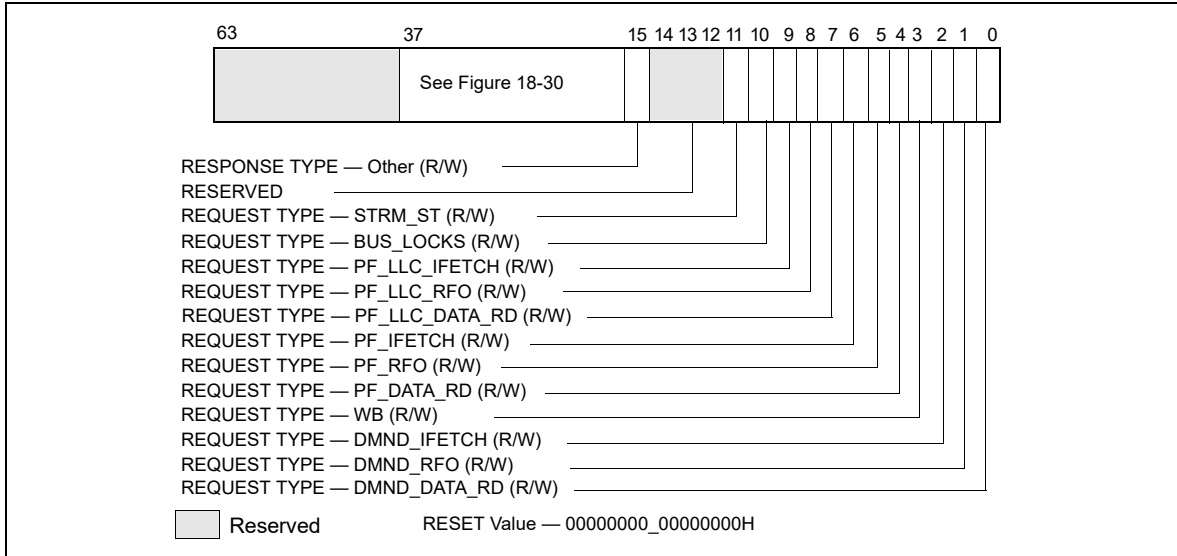


Figure 18-30. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 18-16. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	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	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	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
OTHER	15	Any other request that crosses IDI, including I/O.

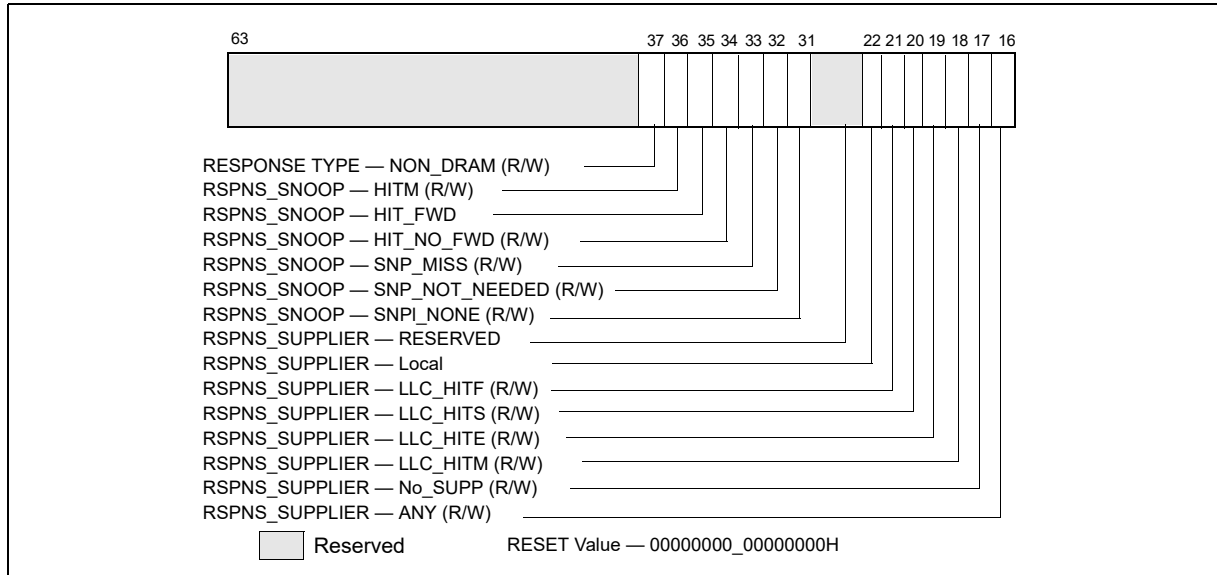


Figure 18-31. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

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 18-17. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	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 18-18. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	SNP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNP_MISS	33	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	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	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	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	Target was non-DRAM system address. This includes MMIO transactions.

18.3.4.6 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 18.3.1.2.2. 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 18-32.

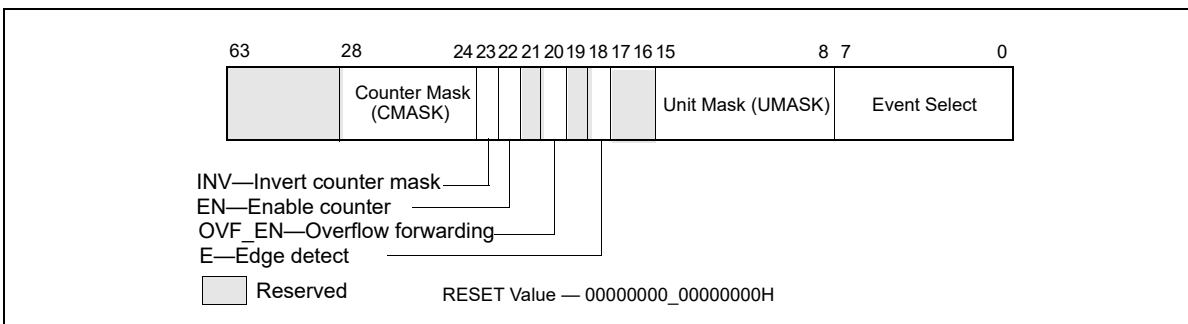


Figure 18-32. Layout of Uncore PERFVTSSEL MSR for a C-Box Unit or the ARB Unit

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 19-20.
- 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 18-33 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).

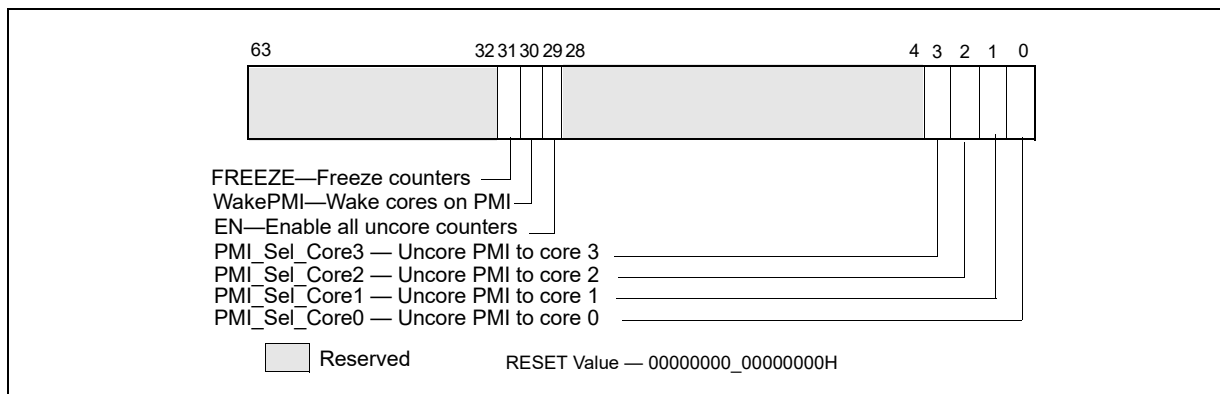


Figure 18-33. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSR for uncore PMU for each box.

Table 18-19. 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, see Table 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

18.3.4.6.1 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 19-20 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 cannot 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.

18.3.4.7 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 18.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 18.6.3 through Section 18.3.4.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 18-17 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 18-20. Additionally, there are some small differences in the non-architectural performance monitoring events (see Table 19-18).

Table 18-20. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Remote	30:23	Remote DRAM Controller (either all 0s or all 1s).

18.3.4.8 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 18-21 summarizes the uncore PMU facilities providing MSR interfaces.

Table 18-21. 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 2-24.

18.3.5 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 18.6.3 through Section 18.3.4.5. The non-architectural performance monitoring events supported by the processor core are listed in Table 19-18.

18.3.5.1 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 2-28.

18.3.6 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 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. 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 18.3.6.5. 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 18-22. Core PMU Comparison

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
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 determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM. 	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM. 	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12 and Section 18.3.6.5.1.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 18.3.4.4.3	
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 17.11.	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 18.3.6.5.	No	

18.3.6.1 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 micro-architecture 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 18-23.

Table 18-23. 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 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-29	
PEBS record layout	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 18-3; enhanced fields at offsets 98H, A0H, A8H.	
Precise Events	See Table 18-12.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-13	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 18.3.4.4.3.	
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.

18.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 18-24. 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 18-24. 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 18.3.6.5.1)

The layout of PEBS records are almost identical to those shown in Table 18-3. 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 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 18.3.6.3).

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.

18.3.6.3 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 18-25. 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 18-25. 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 an event listed in Table 18-25 using any one of IA32_PERFVTSEL0-IA32_PERFVTSEL3.
- 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 18-26.

Table 18-26. 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	<ul style="list-style-type: none"> ▪ 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 18-25.
Reserved	A8H	Always zero.

18.3.6.3.1 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.

18.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. The event codes are listed in Table 18-15. 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 18-27.
- Supplier information (bits 30:16): see Table 18-28.
- Snoop response information (bits 37:31): see Table 18-18.

Table 18-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell microarchitecture)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
COREWB	3	Counts the number of modified cachelines written back.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_LOCK	10	Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	Counts the number of streaming store requests electronically.
Reserved	14:12	Reserved
OTHER	15	Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-28. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 18-28. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_3CH, 06_46H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

Table 18-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	L4 Cache
	Reserved	30:26	Reserved

18.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 18-28 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).

Table 18-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	L3_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

18.3.6.5 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 its 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 18-34. 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 2-29.

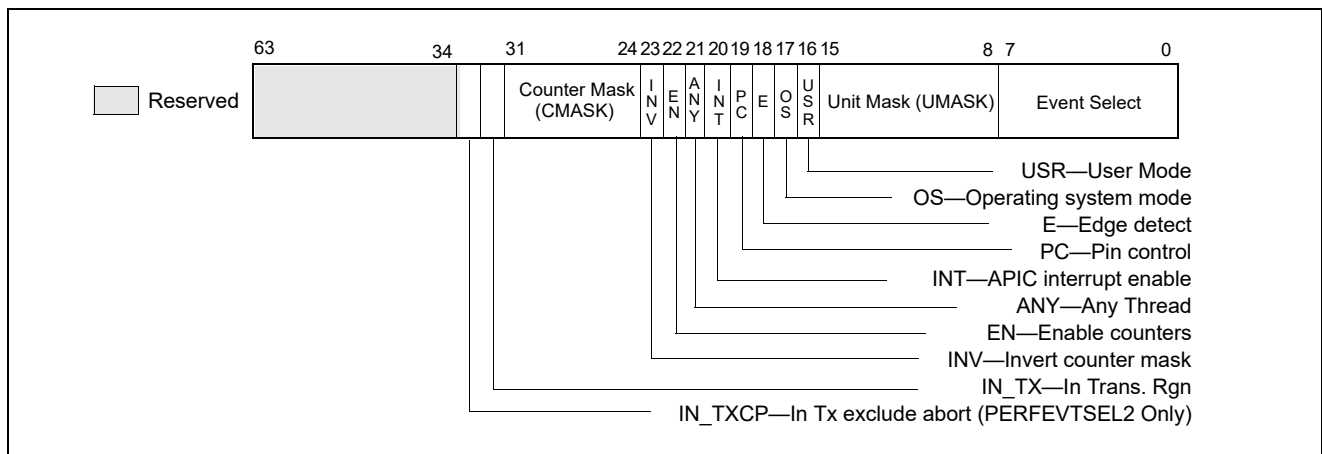


Figure 18-34. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

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.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they are listed in Table 19-12.

18.3.6.5.1 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 19-12 also support using PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORTED (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 pending 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 18-31.

Table 18-31. 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

18.3.6.6 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 18.3.4.6.

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 18-32.

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 18-33 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs for uncore PMU for each box.

Table 18-32. 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, see Table 2-21 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 19-13.

18.3.6.7 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 2-33.

18.3.7 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 18.3.6. 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.

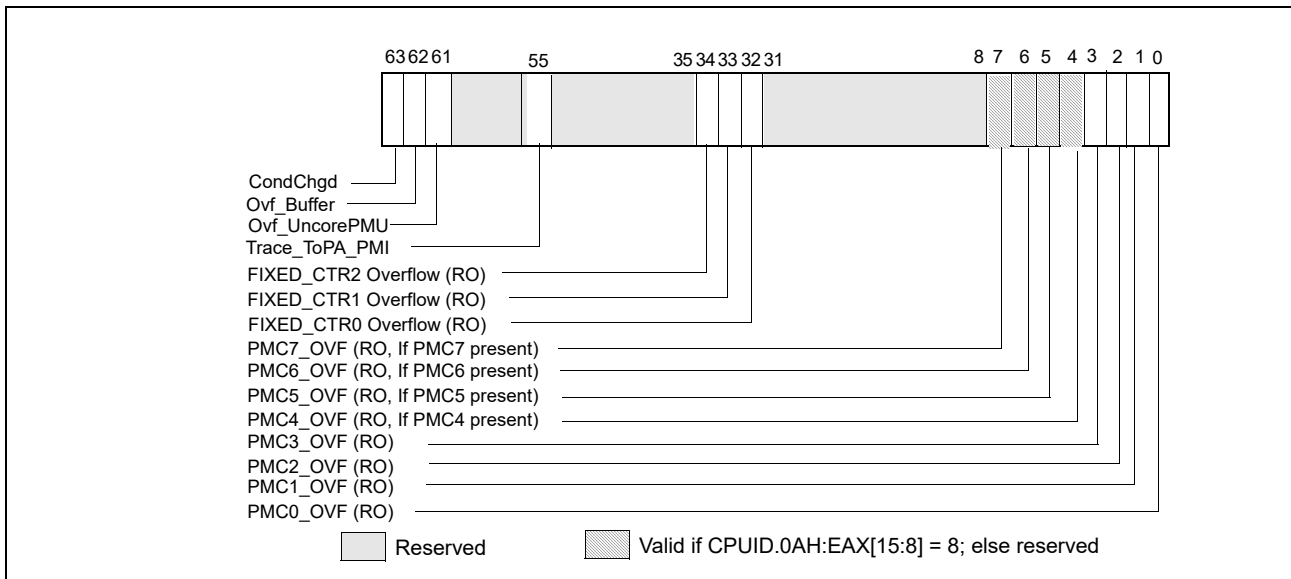


Figure 18-35. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

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.

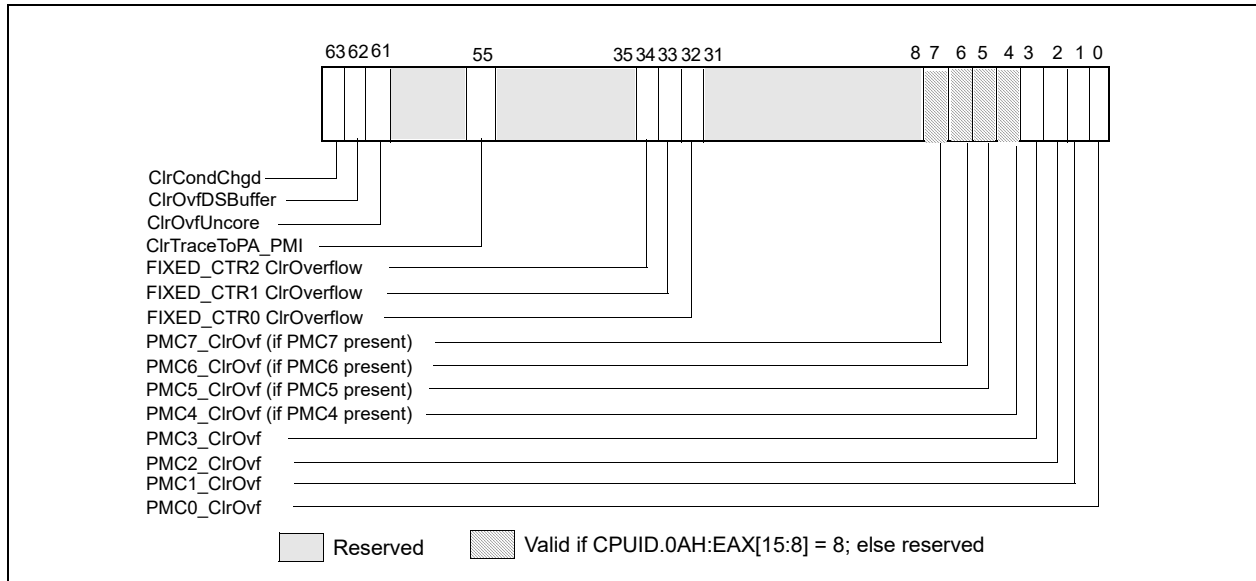


Figure 18-36. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, “Performance Monitoring Events”.

18.3.8 6th Generation, 7th Generation and 8th 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 8th generation Intel® Core™ processor is based on the Coffee Lake microarchitecture. For these microarchitectures, 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 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. 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 18.3.6.5. 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 18-33. Core PMU Comparison

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee Lake	Intel® Microarchitecture Code Name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 18.2.1.
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 determine # of counters. See Section 18.2.1.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics. ▪ Freeze_while_SMM. 	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI 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	<ul style="list-style-type: none"> ▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET 	<ul style="list-style-type: none"> ▪ 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	<ul style="list-style-type: none"> ▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI 	<ul style="list-style-type: none"> ▪ 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	<ul style="list-style-type: none"> ▪ 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 18-36.	See Table 18-12.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.3.8.1.4.	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 17.12
LBR Timing	Yes	No	Section 17.12.1
Call Stack Profiling	Yes, see Section 17.11	Yes, see Section 17.11	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 18.3.6.5.	See Section 18.3.6.5.	

18.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th 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 18-34.

Table 18-34. PEBS Facility Comparison

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee 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 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-15	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 18-35; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 18-24; 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 18-36.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
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.

18.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 18-35. 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 18-35. PEBS Record Format for 6th Generation, 7th Generation and 8th 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 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-24.

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 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and data address profiling (Section 18.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation 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 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.

18.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake and Coffee Lake microarchitectures is shown in Table 18-36.

Table 18-36. Precise Events for the Skylake, Kaby Lake and Coffee Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRE	C0H	PREC_DIST ¹	01H
		ALL_CYCLES ²	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRE	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRE	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRE	C6H	<Programmable ³ >	01H
HLE_RETIRE	C8H	ABORTED	04H
RTM_RETIRE	C9H	ABORTED	04H
MEM_INST_RETIRE ²	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRE 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_RETIRE ²	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRE.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBBS_FRONTEND, see Section 18.3.8.2
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

18.3.8.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the 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 18-26.

Table 18-37. 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	AOH	<ul style="list-style-type: none"> ▪ 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.

18.3.8.1.4 PEBS Facility for Front End Events

In the 6th generation, 7th generation and 8th 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 front-end 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 18-38.
- 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 18-38.

Table 18-38. 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.
L1L_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 L1L cache miss will not be counted.
L2L_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	<p>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.</p> <p>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.</p>

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 18-39.

Table 18-39. 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 18-38.
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

18.3.8.1.5 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) front-end 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 front-end (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.

18.3.8.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. 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 18-40.
- Supplier information (bits 29:16): see Table 18-41.
- Snoop response information (bits 37:30): see Table 18-42.

Table 18-40. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake, Kaby Lake and Coffee Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	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	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	14:3	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 18-41 lists the supplier information field that applies to 6th generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06_4EH, 06_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06_8EH, 06_9EH).

Table 18-41. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures 06_4EH, 06_5EH and 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT	22	L4 Cache (if L4 is present in the processor).
	Reserved	25:23	Reserved
	DRAM	26	Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).

Table 18-42 lists the snoop information field that apply to processors with CPUID signatures 06_4EH, 06_5EH, 06_8EH, 06_9E, and 06_55H.

**Table 18-42. MSR_OFFCORE_RSP_x Snoop Info Field Definition
(CPUID Signatures 06_4EH, 06_5EH, 06_8EH, 06_9E and 06_55H)**

Subtype	Bit Name	Offset	Description
Snoop Info	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).
	SNOOP_NONE	31	No details on snoop-related information.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	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	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	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	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	Target was non-DRAM system address. This includes MMIO transactions.	

18.3.8.2.1 Off-core Response Performance Monitoring for the Intel® Xeon® Processor Scalable Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Processor Scalable Family.

- Transaction request type encoding (bits 15:0): see Table 18-43.
- Supplier information (bits 29:16): see Table 18-44.
- Supplier information (bits 29:16) with support for Intel® Optane™ DC Persistent Memory support: see Table 18-45.
- Snoop response information has not been changed and is the same as in (bits 37:30): see Table 18-42.

Table 18-43. MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Processor Scalable Family)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
Reserved	14:11	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 18-44 lists the supplier information field that applies to the Intel Xeon Processor Scalable Family (CPUID signature: 06_55H).

Table 18-44. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_55H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	Reserved	25:22	Reserved
	L3_MISS_LOCAL_DRAM	26	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
Reserved	30	Reserved	

Table 18-45 lists the supplier information field that applies to the Intel Xeon Processor Scalable Family (CPUID signature: 06_55H, Steppings 0x5H - 0xFH).

**Table 18-45. MSR_OFFCORE_RSP_x Supplier Info Field Definition
(CPUID Signature 06_55H, Steppings 0x5H - 0xFH)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	LOCAL_PMM	22	Local home requests that were serviced by local PMM.
	REMOTE_HOP0_PMM	23	Hop 0 Remote supplier.
	REMOTE_HOP1_PMM	24	Hop 1 Remote supplier.
	REMOTE_HOP2P_PMM	25	Hop 2 or more Remote supplier.
	L3_MISS_LOCAL_DRAM	26	L3 Miss: Local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
Reserved		30	Reserved

18.3.8.3 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Cannon Lake microarchitecture introduces LLC support of up to six processor cores. To support six processor cores and eight LLC slices, existing MSRs have been rearranged and new CBo MSRs have been added. Uncore performance monitoring software drivers from prior generations of Intel Core processors will need to update the MSR addresses. The new MSRs and updated MSR addresses have been added to the Uncore PMU listing in Section 2.17.2, “MSRs Specific to 8th Generation Intel® Core™ i3 Processors” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

18.3.9 Next Generation Intel® Core™ Processor Performance Monitoring Facility

The next generation Intel® Core™ processor is based on Ice Lake microarchitecture. For this microarchitecture, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 18.2.5) and a host of non-architectural monitoring capabilities.

The core PMU's capability is similar to those described in Section 18.3.1 through Section 18.3.8, with some differences and enhancements summarized in Table 18-46.

Table 18-46. PEBS Facility Comparison

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Architectural Perfmon version	5	4	See Section 18.2.5.
PEBS: Basic functionality	Yes	Yes	See Section 18.3.9.1.
PEBS record format encoding	0100b	0011b	See Section 18.6.2.4.2.

Table 18-46. PEBS Facility Comparison

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Extended PEBS	PEBS is extended to all Fixed and General Purpose counters and to all performance monitoring events.	No	See Section 18.9.1.
Adaptive PEBS	Yes	No	See Section 18.9.2.
Performance Metrics	Yes (4)	No	See Section 18.3.9.3.
PEBS-PDIR	IA32_FIXED0 only (Corresponding counter control MSRs must be enabled.)	IA32_PMC1 only.	

18.3.9.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the next generation Intel Core processors provides a number of enhancements relative to PEBS in processors based on the Skylake, Kaby Lake, and Coffee Lake microarchitectures. Enhancement of PEBS facility with Extended PEBS and Adaptive PEBS features are described in detail in Section 18.9.

18.3.9.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. 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-[N1].
- Response type encoding (bits 16-37) of
 - Supplier information: see Table [18-N2].
 - Snoop response information: see Table [18-N3].
- All transactions are tracked at cacheline granularity except some in request type OTHER.

**Table 18-47. MSR_OFFCORE_RSP_x Request_Type Definition
(Future Processors Based on Ice Lake Microarchitecture)**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data and page table entry reads.
DEMAND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DEMAND_CODE_RD	2	Counts demand instruction fetches and instruction prefetches targeting the L1 instruction cache.
Reserved	3	Reserved
HWPf_L2_DATA_RD	4	Counts hardware generated data read prefetches targeting the L2 cache.
HWPf_L2_RFO	5	Counts hardware generated prefetches for exclusive ownership (RFO) targeting the L2 cache.
Reserved	6	Reserved
HWPf_L3	9:7 and 13 ¹	Counts hardware generated prefetches of any type targeting the L3 cache.
HWPf_L1D_AND_SWPF	10	Counts hardware generated data read prefetches targeting the L1 data cache and the following software prefetches (PREFETCHNTA, PREFETCHT0/1/2).
STREAMING_WR	11	Counts streaming stores.
Reserved	12	Reserved
Reserved	14	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

NOTES:

1. All bits need to be set to 1 to count this type.

Ice Lake microarchitecture has added a new category of Response subtype, called a Combined Response Info. To count a feature in this type, all the bits specified must be set to 1.

A valid response type must be a non-zero value of the following expression:

Any | ['OR' of Combined Response Info Bits | (('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit[16] is set, other response type bits [17-39] are ignored.

Table 18-48 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

**Table 18-48. MSR_OFFCORE_RSP_x Supplier Info Field Definition
(Future Processors Based on Ice Lake Microarchitecture)**

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Combined Response Info	DRAM	26, 31, 32 ¹	Requests that are satisfied by DRAM.
	NON_DRAM	26, 37 ¹	Requests that are satisfied by a NON_DRAM system component. This includes MMIO transactions.
	L3_MISS	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37 ¹	Requests that were not supplied by the L3 Cache. The event includes some currently reserved bits in anticipation of future memory designs.
Supplier Info	L3_HIT	18,19, 20 ¹	Requests that hit in L3 cache. Depending on the snoop response the L3 cache may have retrieved the cacheline from another core's cache.
Reserved		17, 21:25, 27:29	Reserved.

NOTES:

1. All bits need to be set to 1 to count this type.

Table 18-49 lists the snoop information field that applies to processors based on Ice Lake microarchitecture.

**Table 18-49. MSR_OFFCORE_RSP_x Snoop Info Field Definition
(Future Processors Based on Ice Lake Microarchitecture)**

Subtype	Bit Name	Offset	Description
Snoop Info	Reserved	30	Reserved.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was sent and none of the snooped caches contained the cacheline.
	SNOOP_HIT_NO_FWD	34	A snoop was sent and hit in at least one snooped cache. The unmodified cacheline was not forwarded back, because the L3 already has a valid copy.
	Reserved	35	Reserved.
	SNOOP_HITM	36	A snoop was sent and the cacheline was found modified in another core's caches. The modified cacheline was forwarded to the requesting core.

18.3.9.3 Performance Metrics

The Ice Lake core PMU provides built-in support for Top-down Microarchitecture Analysis (TMA) method level 1 metrics. These metrics are always available to cross-validate performance observations, freeing general purpose counters to count other events in high counter utilization scenarios. For more details about the method, refer to Top-Down Analysis Method chapter (Appendix B.1) of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A new MSR called MSR_PERF_METRICS reports the metrics directly. Software can check (and/or expose to its guests) the availability of the PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15).

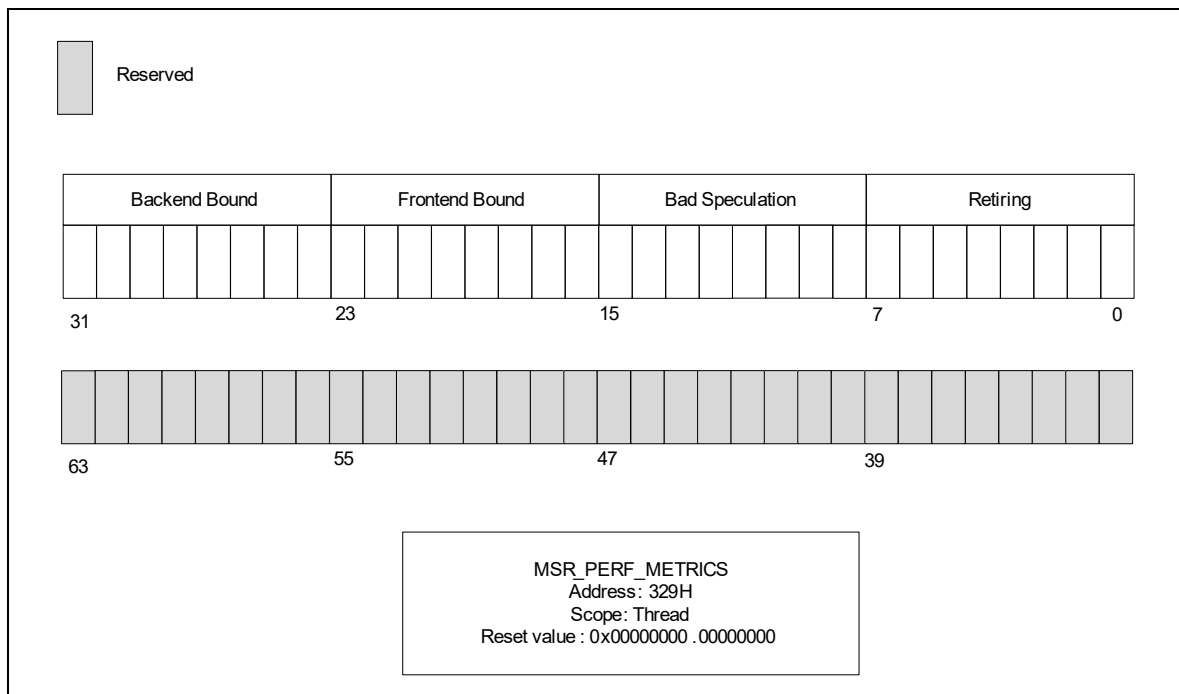


Figure 18-37. MSR_PERF_METRICS Definition

This register exposes the four TMA Level 1 metrics. The lower 32 bits are divided into four 8-bit fields, as shown by the above figure, each of which is an integer fraction of 255.

To support built-in performance metrics, new bits have been added to the following MSRs:

- IA32_PERF_GLOBAL_CTRL. EN_PERF_METRICS[48]: If this bit is set and fixed counter 3 is enabled, built-in performance metrics are enabled.
- IA32_PERF_GLOBAL_STATUS_SET. SET_OVF_PERF_METRICS[48]: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS_RESET. RESET_OVF_PERF_METRICS[48]: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS. OVF_PERF_METRICS[48]: If this bit is set, it indicates that a PERF_METRICS-related resource has overflowed and a PMI is triggered¹. If this bit is clear, no such overflow has occurred.

NOTE

Software has to synchronize, e.g., re-start, fixed counter 3 as well as PERF_METRICS when either bit 35 or 48 in IA32_PERF_GLOBAL_STATUS is set. Otherwise, PERF_METRICS may return undefined values.

The values in MSR_PERF_METRICS are derived from fixed counter 3. Software should start both registers, PERF_METRICS and fixed counter 3, from zero. Additionally, software is recommended to periodically clear both registers in order to maintain accurate measurements for certain scenarios that involve sampling metrics at high rates.

In order to save/restore PERF_METRICS, software should follow these guidelines:

- PERF_METRICS and fixed counter 3 should be saved and restored together.
- To ensure that PERF_METRICS and fixed counter 3 remain synchronized, both should be disabled during both save and restore. Software should enable/disable them atomically, with a single write to IA32_PERF_GLOBAL_CTRL to set/clear both EN_PERF_METRICS[bit 48] and EN_FIXED_CTR3[bit 35].
- On state restore, fixed counter 3 must be restored **before** PERF_METRICS, otherwise undefined results may be observed.

18.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

NOTE

This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

18.4.1 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.

1. An overflow of fixed counter 3 should normally happen first if software follows Intel's recommendations.

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. The processor supports AnyThread counting in three architectural performance monitoring events.

18.4.1.1 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.

18.4.1.1.1 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 18.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

Table 18-50. 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 18-50. 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 18-51, and each field in the PEBS record is 64 bits long.

Table 18-51. 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

18.4.1.1.2 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 18-52 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 18-52. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-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 18-53. 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 18.5.2.3 for details.

Table 18-53. 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.
Response Type	Any	15	ANY_REQUEST	Account for any requests.
	Data Supply from Untile	16	ANY_RESPONSE	Account for any response.
		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.
		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.

Table 18-53. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
	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 PMCO. 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 PMCO. This bit is reserved for OFFCORE_RESP_1 event.

18.4.1.1.3 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 18.5.2.3, "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.

18.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS)

18.5.1 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).

NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 18.2.1.

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 19-31.

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 18-71, Table 18-72, Table 18-73, and Table 18-74. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-31 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").

18.5.2 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 19-30.

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.

18.5.2.1 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.

18.5.2.1.1 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 18.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 18-54.

Table 18-54. 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 18-55, and each field in the PEBS record is 64 bits long.

Table 18-55. 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

Table 18-55. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

18.5.2.2 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 18-56 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 18-56. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 18-38 and Figure 18-39. 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 18.5.2.3 for details.

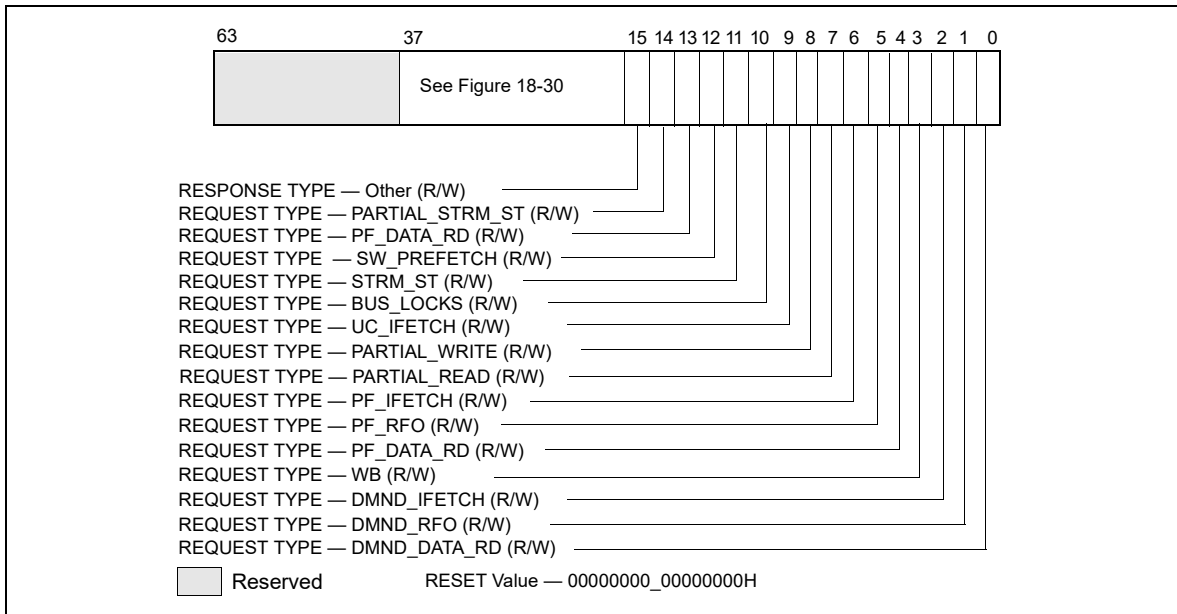


Figure 18-38. Request_Type Fields for MSR_OFFCORE_RSPx

Table 18-57. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	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	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	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	Counts the number of UC instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
SW_PREFETCH	12	Counts software prefetch requests
PF_DATA_RD	13	Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	Streaming store requests
ANY	15	Any request that crosses IDI, including I/O.

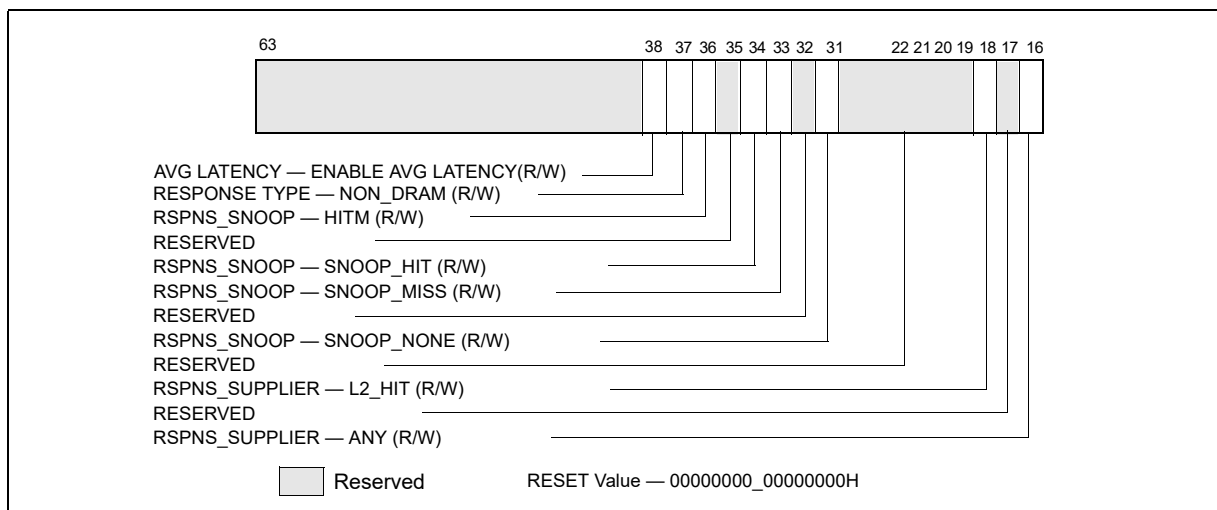


Figure 18-39. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 18-57) and a valid response type pattern (Table 18-58, Table 18-59). 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 18-58. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	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 18-59. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved
	HITM	36	Counts the number of snoops hit in the other module where modified copies were found in other core’s L1 cache.
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	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.

18.5.2.3 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_PERFVTSELx 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, while 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.

18.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture 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. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 18.5.2, with some differences and enhancements summarized in Table 18-60.

Table 18-60. 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 determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	2	Use CPUID to determine # of counters. See Section 18.2.1.
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 determine # of counters. See Section 18.2.1.
PMI Overhead Mitigation	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling. 	<ul style="list-style-type: none"> ▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. 	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> ▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_R ESET ▪ Set via IA32_PERF_GLOBAL_STATUS_S ET 	<ul style="list-style-type: none"> ▪ 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	<ul style="list-style-type: none"> ▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz 	<ul style="list-style-type: none"> ▪ Individual counter overflow ▪ PEBS buffer overflow 	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> ▪ 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 18-61.	See Section 18.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-54).	IA32_PMC0 only.

Table 18-60. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 18-62; enhanced fields at offsets 90H- 98H; and TSC record field at COH.	Table 18-55.	
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.

18.5.3.1 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 on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 18.6.2.4 and Section 17.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 18-61.

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. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. CPU_CLK_UNHALTED.CORE_P will increment each cycle that the processor is awake. When the counter overflows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, 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 18-61. 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_RETIRE	C0H	ANY	00H
UOPS_RETIRE	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRE	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_RETIRE	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRE	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRE	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 Goldmont microarchitecture is shown in Table 18-62, and each field in the PEBS record is 64 bits long.

Table 18-62. 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 indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

18.5.3.1.1 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; those fields are present in the record but are reserved.

For Goldmont microarchitecture, 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). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

18.5.3.1.2 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 18.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRE, except for UOPS_RETIRE. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

With Reduced Skid PEBS, the skid is precisely one event occurrence. Hence if counting INST_RETIRE, PEBS will indicate the instruction that follows that which caused the counter to overflow.

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).

18.5.3.1.3 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.

18.5.3.2 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 18-56 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 18-63.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-58.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-64.
- 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 18.5.2.3 for details.

Table 18-63. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	Counts cacheline requests due to software prefetch instructions.

Table 18-63. MSR_OFFCORE_RSPx Request_Type Field Definition (Contd.)

Bit Name	Offset	Description
PF_L1_DATA_RD	13	Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-57) and a valid response type pattern (either Table 18-58 or Table 18-64). 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 18-64. 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_0 R_NO_SNOOP_NEEDED	33	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_CO RE_NO_FWD	34	A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_C ORE	36	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	Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests ¹	OUTSTANDING	38	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_RESPO.

NOTES:

1. See Section 18.5.2.3, "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_Response Bit | L2 Hit | 'OR' of Snoop Info Bits | Outstanding Bit

18.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 18.5.2.3.

18.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 18.6.3, with some differences and enhancements summarized in Table 18-65.

Table 18-65. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change.
Architectural Performance Monitoring version ID	4	4	No change.
Processor Event Based Sampling (PEBS) Events	All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-61.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

18.5.4.1 Extended PEBS

The PEBS facility in Goldmont Plus microarchitecture provides a number of enhancements relative to PEBS in processors from previous generations. Enhancement of PEBS facility with the Extended PEBS feature are described in detail in section 18.9.

18.5.5 Performance Monitoring for Tremont Microarchitecture

Intel Atom processors based on the Tremont microarchitecture report architectural performance monitoring versionID = 5 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 5 capabilities are described in Section 18.2.5.

Tremont performance monitoring capabilities are similar to Goldmont Plus capabilities, with the following extensions:

- Support for Adaptive PEBS.
- Support for PEBS output to Intel® Processor Trace.
- Precise Distribution support on Fixed Counter0.
- Compatibility enhancements to off-core response MSRs, MSR_OFFCORE_RSPx.

The differences and enhancements between Tremont microarchitecture and Goldmont Plus microarchitecture are summarized in Table 18-66.

Table 18-66. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures

Box	Tremont Microarchitecture	Goldmont Plus Microarchitecture	Comment
# of fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 18.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 18.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change. See Section 18.2.2.
Architectural Performance Monitoring version ID	5	4	
PEBS record format encoding	0100b	0011b	See Section 18.6.2.4.2.
Reduce skid PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 only	
Extended PEBS	Yes	Yes	See Section 18.5.4.1.
Adaptive PEBS	Yes	No	See Section 18.9.2.
PEBS output	DS Save Area or Intel® Processor Trace	DS Save Area only	See Section 18.5.5.2.1.
PEBS record layout	See Section 18.9.2.3 for output to DS, Section 18.5.5.2.2 for output to Intel PT.	Table 18-62; enhanced fields at offsets 90H- 98H; and TSC record field at COH.	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register.	

18.5.5.1 Adaptive PEBS

The PEBS record format and configuration interface has changed versus Goldmont Plus, as the Tremont microarchitecture includes support for the configurable Adaptive PEBS records; see Section 18.9.2.

18.5.5.2 PEBS output to Intel® Processor Trace

Intel Atom processors based on the Tremont microarchitecture introduce the following Precise Event-Based Sampling (PEBS) extensions:

- A mechanism to direct PEBS output into the Intel® Processor Trace (Intel® PT) output stream. In this scenario, the PEBS record is written in packetized form, in order to co-exist with other Intel PT trace data.
- New Performance Monitoring counter reload MSRs, which are used by PEBS in place of the counter reload values stored in the DS Management area when PEBS output is directed into the Intel PT output stream.

Processors that indicate support for Intel PT by setting CPUID.07H.0.EBX[25]=1, and set the new IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] bit, support these extensions.

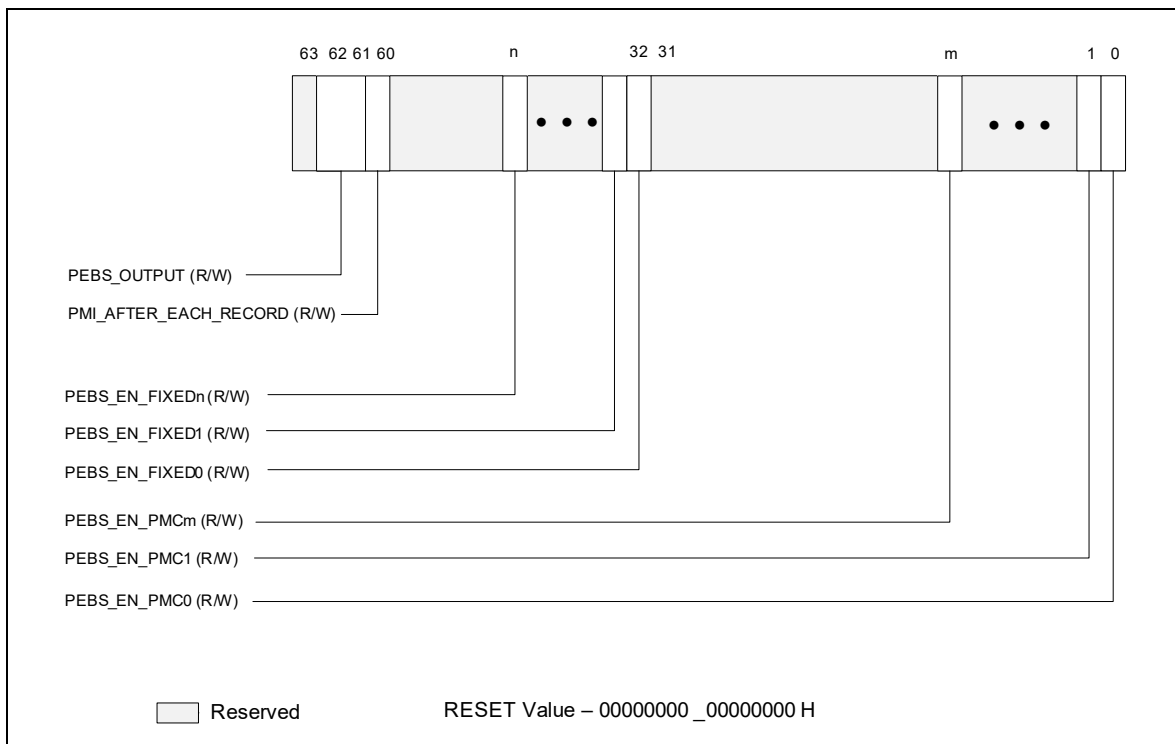
18.5.5.2.1 PEBS Configuration

PEBS output to Intel Processor Trace includes support for two new fields in IA32_PEBS_ENABLE.

Table 18-67. New Fields in IA32_PEBS_ENABLE

Field	Description
PMI_AFTER_EACH_RECORD[60]	Pend a PerfMon Interrupt (PMI) after each PEBS event.
PEBS_OUTPUT[62:61]	Specifies PEBS output destination. Encodings: 00B: DS Save Area. Matches legacy PEBS behavior, output location defined by IA32_DS_AREA. 01B: Intel PT trace output. 10B: Reserved. 11B: Reserved.

When PEBS_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32_RTIT_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR_RELOAD_PMCx MSRs. Details on configuring Intel PT can be found in Section 35.2.6.

**Figure 18-40. IA32_PEBS_ENABLE MSR with PEBS Output to Intel® Processor Trace**

18.5.5.2.2 PEBS Record Format in Intel® Processor Trace

The format of the PEBS record changes when output to Intel PT, as the PEBS state is packetized. Each PEBS grouping is emitted as a Block Begin (BBP) and following Block Item (BIP) packets. A PEBS grouping ends when either a new PEBS grouping begins (indicated by a BBP packet) or a Block End (BEP) packet is encountered. See Section 35.4.1.1 for details of these Intel PT packets.

Because the packet headers describe the state held in the packet payload, PEBS state ordering is not fixed. PEBS state groupings may be emitted in any order, and the PEBS state elements within those groupings may be emitted in any order. Further, there is no packet that provides indication of "Record Format" or "Record Size".

If Intel PT tracing is not enabled (IA32_RTIT_STATUS.TriggerEn=0), any PEBS records triggered will be dropped. PEBS packets do not depend on ContextEn or FilterEn in IA32_RTIT_STATUS, any filtering of PEBS must be enabled from within the PerfMon configuration. Counter reload will occur in all scenarios where PEBS is triggered, regardless of TriggerEn.

The PEBS threshold mechanism for generating PerfMon Interrupts (PMIs) is not available in this mode. However, there exist other means to generate PMIs based on PEBS output. When the Intel PT ToPA output mechanism is chosen, a PMI can optionally be pended when a ToPA region is filled; see Section 35.2.6.2 for details. Further, software can opt to generate a PMI on each PEBS record by setting the new IA32_PEBS_ENABLE.PMI_AFTER_EACH_RECORD[60] bit.

The IA32_PERF_GLOBAL_STATUS.OvfDSBuffer bit will not be set in this mode.

18.5.5.2.3 PEBS Counter Reload

When PEBS output is directed into Intel PT (IA32_PEBS_ENABLE.PEBS_OUTPUT = 01B), new MSR_RELOAD_PMCx MSRs are used by the PEBS routine to reload PerfMon counters. The value from the associated reload MSR will be loaded to the appropriate counter on each PEBS event.

18.5.5.3 Precise Distribution Support on Fixed Counter 0

The Tremont microarchitecture supports the PDIR (Precise Distribution of Retired Instructions) facility, as described in Section 18.3.4.4.4, on Fixed Counter 0. Fixed Counter 0 counts the INST_RETIRED.ALL event. PEBS skid for Fixed Counter 0 will be precisely one instruction.

This is in addition to the reduced skid PEBS behavior on IA32_PMC0; see Section 18.5.3.1.2.

18.5.5.4 Compatibility Enhancements to Offcore Response MSRs

The Off-core Response facility is similar to that described in Section 18.5.3.2.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as shown below. RequestType bits are defined in Table 18-68, ResponseType bits in Table 18-69, and SnoopInfo bits in Table 18-70.

Table 18-68. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPf_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPf_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPf_L2_CODE_RD	6	Counts all prefetch (that bring data to L2 only) code reads.
Reserved	9:7	Reserved.
HWPf_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
Reserved	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (WiL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (WCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (WCiLF).

Table 18-68. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

Table 18-69. MSR_OFFCORE_RSPx Response Type Definition

Bit Name	Offset	Description
ANY_RESPONSE	16	Catch all value for any response types.
L3_HIT_M	18	LLC/L3 Hit - M-state.
L3_HIT_E	19	LLC/L3 Hit - E-state.
L3_HIT_S	20	LLC/L3 Hit - S-state.
L3_HIT_F	21	LLC/L3 Hit - I-state.
LOCAL_DRAM	26	LLC/L3 Miss, DRAM Hit.
OUTSTANDING	63	Average latency of outstanding requests with the other counter counting number of occurrences; can also can be used to count occupancy.

Table 18-70. MSR_OFFCORE_RSPx Snoop Info Definition

Bit Name	Offset	Description
SNOOP_NONE	31	None of the cores were snooped. <ul style="list-style-type: none"> LLC miss and Dram data returned directly to the core.
SNOOP_NOT_NEEDED	32	No snoop needed to satisfy the request. <ul style="list-style-type: none"> LLC hit and CV bit(s) (core valid) was not set. LLC miss and Dram data returned directly to the core.
SNOOP_MISS	33	A snoop was sent but missed. <ul style="list-style-type: none"> LLC hit and CV bit(s) was set but snoop missed (silent data drop in core), data returned from LLC. LLC miss and Dram data returned directly to the core.
SNOOP_HIT_NO_FWD	34	A snoop was sent but no data forward. <ul style="list-style-type: none"> LLC hit and CV bit(s) was set but no data forward from the core, data returned from LLC. LLC miss and Dram data returned directly to the core.
SNOOP_HIT_WITH_FWD	35	A snoop was sent and non-modified data was forward. <ul style="list-style-type: none"> LLC hit and CV bit(s) was set, non-modified data was forward from core.
SNOOP_HITM	36	A snoop was sent and modified data was forward. <ul style="list-style-type: none"> LLC hit E or M and the CV bit(s) was set, modified data was forward from core.
NON_DRAM_BIT	37	Target was non-DRAM system address, MMIO access. <ul style="list-style-type: none"> LLC miss and Non-Dram data returned.

The Off-core Response capability behaves as follows:

- To specify a complete offcore response filter, software must properly program at least one RequestType and one ResponseType. A valid request type must have at least one bit set in the non-reserved bits of 15:0 or 49:44. A valid response type must be a non-zero value of one the following expressions:
 - Read requests:
Any_Response Bit | ('OR' of Supplier Info Bits) 'AND' ('OR' of Snoop Info Bits) | Outstanding Bit
 - Write requests:
Any_Response Bit | ('OR' of Supplier Info Bits) | Outstanding Bit
- When the ANY_RESPONSE bit in the ResponseType is set, all other response type bits will be ignored.
- True Demand Cacheable Loads include neither L1 Prefetches nor Software Prefetches.
- Bits 15:0 and Bits 49:44 specifies the request type of a transaction request to the uncore. This is described in Table 18-68.
- Bits 30:16 specifies common supplier information.
- "Outstanding Requests" (bit 63) is only available on MSR_OFFCORE_RSP0; a #GP fault will occur if software attempts to write a 1 to this bit in MSR_OFFCORE_RSP1. It is mutually exclusive with any ResponseType. Software must guarantee that all other ResponseType bits are set to 0 when the "Outstanding Requests" bit is set.
- "Outstanding Requests" bit 63 can enable measurement of the average latency of a specific type of off-core transaction; two programmable counters must be used simultaneously and the RequestType programming for MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 must be the same when using this Average Latency feature. See Section 18.5.2.3 for further details.

18.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

18.6.1 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 18-71. 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 18-71. 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 18-72.

Table 18-72. 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 18-73.

Table 18-73. 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 18-74. 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 18-74. 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

18.6.2 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 18.6.2.1). 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 18-71, Table 18-72, Table 18-73, and Table 18-74. 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."

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 18-75.

Table 18-75. 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 18-76.

Table 18-76. 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.

18.6.2.1 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 IA32_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 18-41. Two sub-fields are defined for each control. See Figure 18-41; 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.

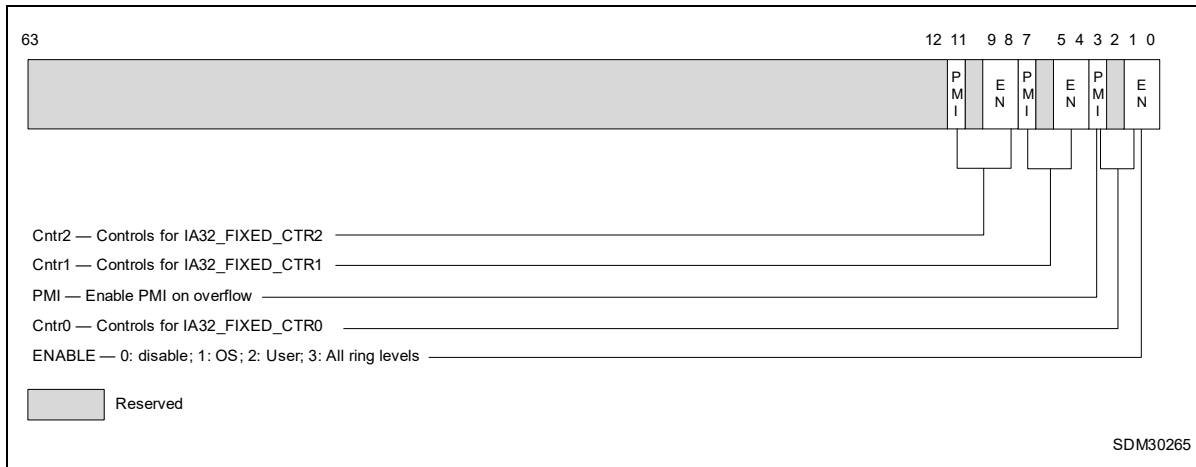


Figure 18-41. Layout of IA32_FIXED_CTR_CTRL MSR

- **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.

18.6.2.2 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 (IA32_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 (IA32_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 (IA32_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 18-42). 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 IA32_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.

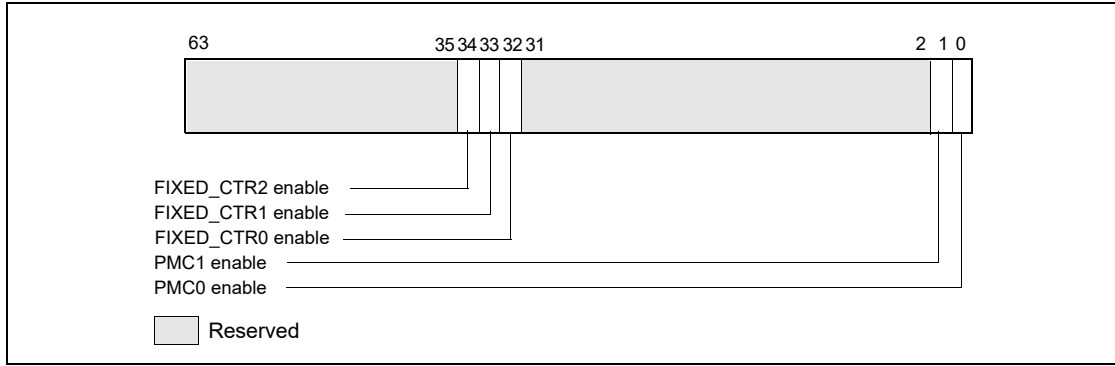


Figure 18-42. Layout of MSR_PERF_GLOBAL_CTRL MSR

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 18-43). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

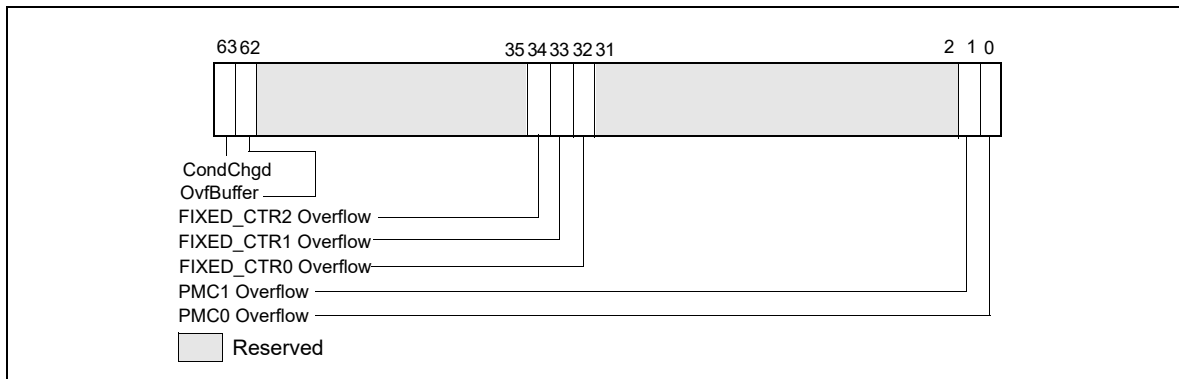


Figure 18-43. Layout of MSR_PERF_GLOBAL_STATUS MSR

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 18-44). 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.

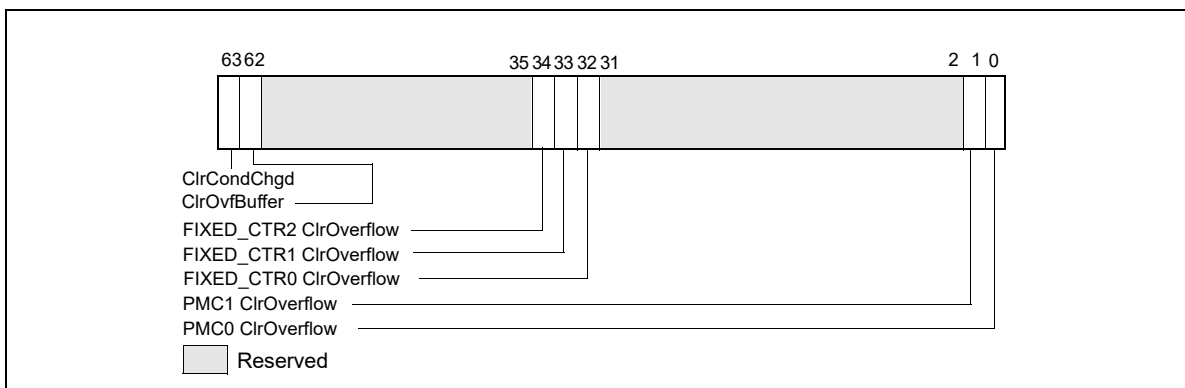


Figure 18-44. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

18.6.2.3 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 18.6.3.6, “At-Retirement Counting”), but is limited to IA32_PMC0. See Table 18-77 for a list of events available to processors based on Intel Core microarchitecture.

Table 18-77. 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

18.6.2.4 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 18.6.2.4.2 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 18-78. The procedure for detecting availability of PEBS is the same as described in Section 18.6.3.8.1.

Table 18-78. 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

18.6.2.4.1 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 18-78.

18.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITIES 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_CAPABILITIES 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:

- **PEBTrap [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 PEBSTrapFormat 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 (See Section 18.6.3.8).
 - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (See Section 18.3.1.1.1).
 - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (See Section 18.3.6.2).
 - 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 18.3.8.1.1).
 - 0100B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 18.9.2.3).

18.6.2.4.3 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 18-79.

Table 18-79. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS.	<ul style="list-style-type: none"> ▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear. 	
Ensure counters are in disabled.	<p>On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0.</p> <p>On subsequent entries:</p> <ul style="list-style-type: none"> ▪ Clear all counters if “Counter Freeze on PMI” is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. <p>Counters MUST be stopped before writing.¹</p>	Optional
Disable PEBS.	Clear ENABLE PMCO 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.	<ul style="list-style-type: none"> ▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable. 	<ul style="list-style-type: none"> ▪ 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 PMCO 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.

18.6.2.4.4 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).

18.6.3 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 18-80 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 18-80. 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_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H

Table 18-80. 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_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
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_SAA_T_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_SAA_T_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_SAA_T_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_SAA_T_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

Table 18-80. 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_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 ¹	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 ¹	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH
					MSR_IQ_COUNTER3	15	30FH
MSR_CRU_ESCR3	5	3CDH					
MSR_CRU_ESCR5	6	3E1H					
MSR_IQ_ESCR1 ¹	0	3BBH					
MSR_RAT_ESCR1	2	3BDH					
MSR_ALF_ESCR1	1	3CBH					
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H			
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 ¹	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 ¹	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	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 19-33) are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events (see Table 19-34) 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 μ ops 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.

18.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-80) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-80) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 18-45 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.)

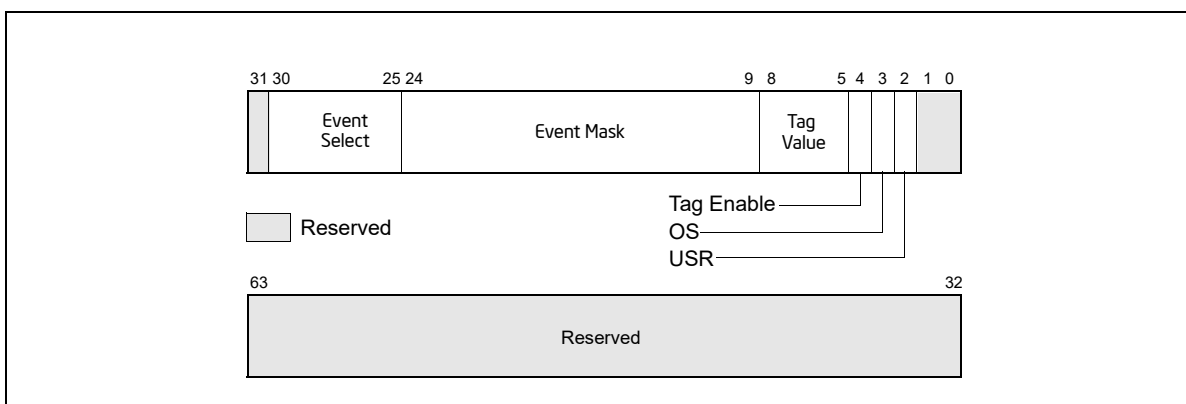


Figure 18-45. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support

- **Tag enable, bit 4** — When set, enables tagging of μ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, "At-Retirement Counting."
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a μ op 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 18-80 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.

18.6.3.2 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 18-80). 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 18.6.3.5.6, "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 18-46). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or, if ECX[31] is set to 1, 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. In such cases, counter bits 31:0 are written to EAX, while 0 is written to EDX.

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.

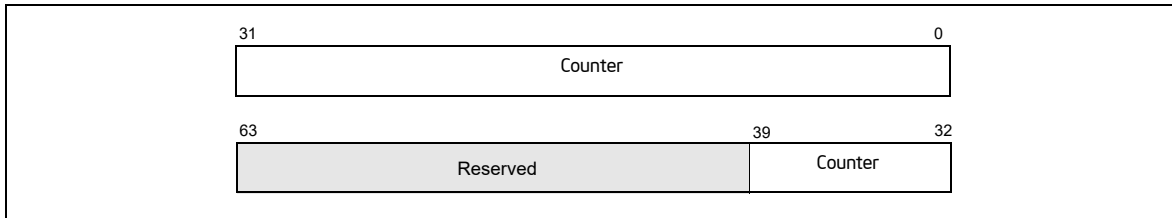


Figure 18-46. Performance Counter (Pentium 4 and Intel Xeon Processors)

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.

18.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 18-80). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 18-47 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 18.6.3.5.2, “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 18.6.3.5.2, “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.

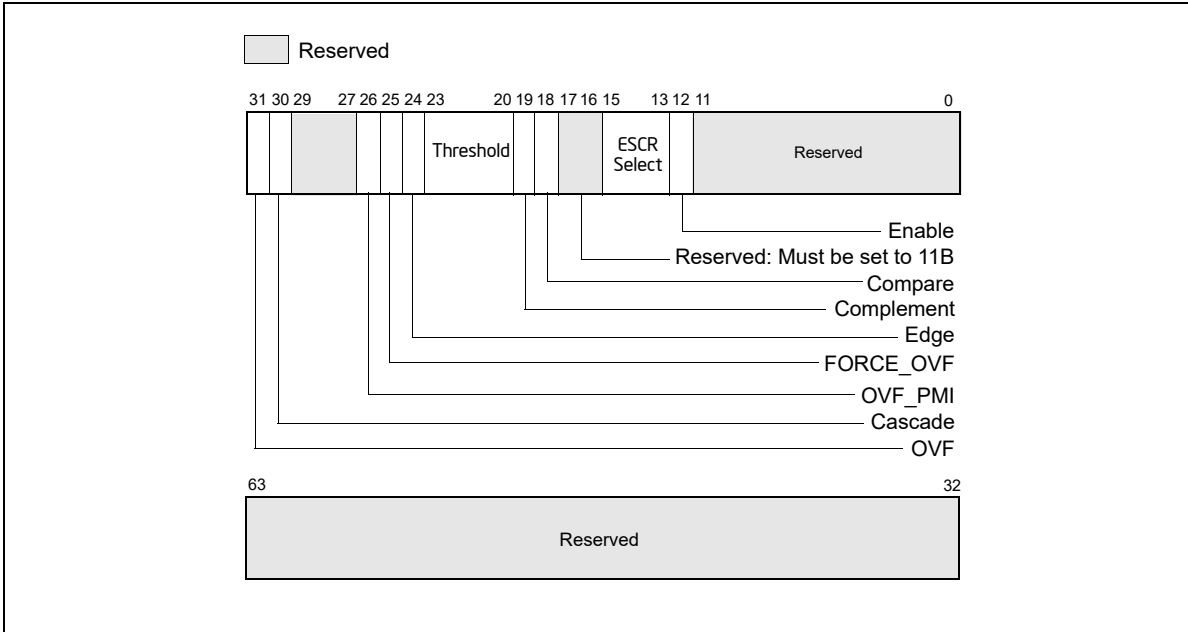


Figure 18-47. Counter Configuration Control Register (CCCR)

- **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 18.6.3.2, “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 18.6.3.5, “Programming the Performance Counters for Non-Retirement Events.”

18.6.3.4 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 18.6.3.8, “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.”

18.6.3.5 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 19-33, Chapter 19.
3. Match the CCCR Select value and ESCR name in Table 19-33 to a value listed in Table 18-80; 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 they 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.

18.6.3.5.1 Selecting Events to Count

Table 19-34 in Chapter 19 lists a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event listed in Table 19-34, setup information is provided. Table 18-81 gives an example of one of the events.

Table 18-81. 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	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]

Table 18-81. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

For Table 19-33 and Table 19-34 in 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 18-80 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 18-80.
- **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 19-34.)
- **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 19-34.)

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 19-33, 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 18-80.
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 18.6.3.5.2, "Filtering Events."

18.6.3.5.2 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 counter's 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, then 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 18-48 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 18.6.3.5.1, "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 18.6.3.5.3, "Starting Event Counting."

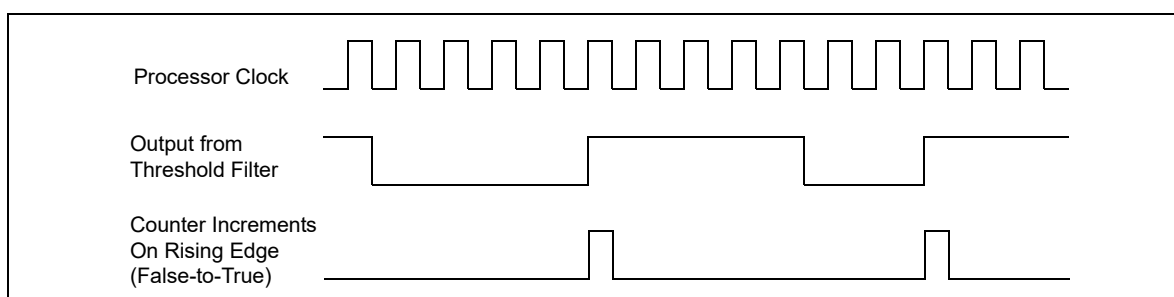


Figure 18-48. Effects of Edge Filtering

18.6.3.5.3 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 18.6.3.5.5, "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 18.6.3.5.2, "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 18.6.3.5.4, "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 18.6.3.5.6, "Cascading Counters").

18.6.3.5.4 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 18.6.3.2, "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 18.6.3.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-80 used as an operand.

This setup procedure is continued in the next section, Section 18.6.3.5.5, "Halting Event Counting."

18.6.3.5.5 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 18.6.3.5.4, "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.

18.6.3.5.6 Cascading Counters

As described in Section 18.6.3.2, "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 18-80). 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 18-47 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 18.6.3.5.1, "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 18.6.3.5.8, "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.

18.6.3.5.7 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 18-82.

Table 18-82. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1 2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INT00	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INT03	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INT04	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INT05	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 CASCNTxINT0y 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 CASCNT4INT00 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFF000H 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.

18.6.3.5.8 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 18.6.3.5.6, "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.

18.6.3.5.9 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).

18.6.3.6 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 19-34 through 19-38 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 μ ops 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 μ ops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μ ops 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 `Ops_Retired` in Table 19-34) can count instructions or μ ops that are retired based on the characterization of bogus” versus non-bogus.
- **Tagging** — Tagging is a means of marking μ ops 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 μ op and a direct count of the event would not provide an indication of how many μ ops encountered that event. The tagging mechanisms allow a μ op 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 μ op, rather than once per event. For example, a μ op may encounter a cache miss more than once during its life time, but a “Miss Retired” metric (that counts the number of retired μ ops that encountered a cache miss) will increment only once for that μ op. 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 μ ops 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, μ ops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μ ops 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 μ ops before they begin and are costly.

18.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μ ops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-34, a μ op 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 μ ops tagged with that mechanism:

- **Front-end tagging** — This mechanism pertains to the tagging of μ ops 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 μ ops that encountered execution events (for example, instruction types) and are counted with the `Execution_Event` event.

- **Replay tagging** — This mechanism pertains to tagging of μ ops 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 μ op that has been tagged using one mechanism will not be detected with another mechanism's tagged- μ op detector. For example, if μ ops are tagged using the front-end tagging mechanisms, the `Replay_event` will not count those as tagged μ ops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μ ops 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 μ ops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

Table 19-34 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 μ op and count tagged μ ops.

18.6.3.6.2 Tagging Mechanism for `Front_end_event`

The `Front_end_event` counts μ ops that have been tagged as encountering any of the following events:

- **μ op decode events** — Tagging μ ops for μ op decode events requires specifying bits in the `ESCR` associated with the performance-monitoring event, `Uop_type`.
- **Trace cache events** — Tagging μ ops for trace cache events may require specifying certain bits in the `MSR_TC_PRECISE_EVENT` MSR (see Table 19-36).

Table 19-34 describes the `Front_end_event` and Table 19-36 describes metrics that are used to set up a `Front_end_event` count.

The MSRs specified in the Table 19-34 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.

18.6.3.6.3 Tagging Mechanism For `Execution_event`

Table 19-34 describes the `Execution_event` and Table 19-37 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 μ ops 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 μ op. 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 μ ops. 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 19-37 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 18.6.3.8.3, "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.

18.6.3.7 Tagging Mechanism for Replay_event

Table 19-34 describes the `Replay_event` and Table 19-38 describes metrics that are used to set up an `Replay_event` count.

The replay mechanism enables tagging of μ ops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μ op 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 19-38 lists the metrics that 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 19-34) be used to count the tagged μ ops.

18.6.3.8 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_PERFVTSEL0` MSRs (See Section 18.6.2.4).

18.6.3.8.1 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.

18.6.3.8.2 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.

18.6.3.8.3 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 19-34 through 19-38.

18.6.3.8.4 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.

18.6.3.8.5 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.

18.6.3.9 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.

18.6.4 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 18.6.3. 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.

18.6.4.1 ESCR MSRs

Figure 18-49 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.

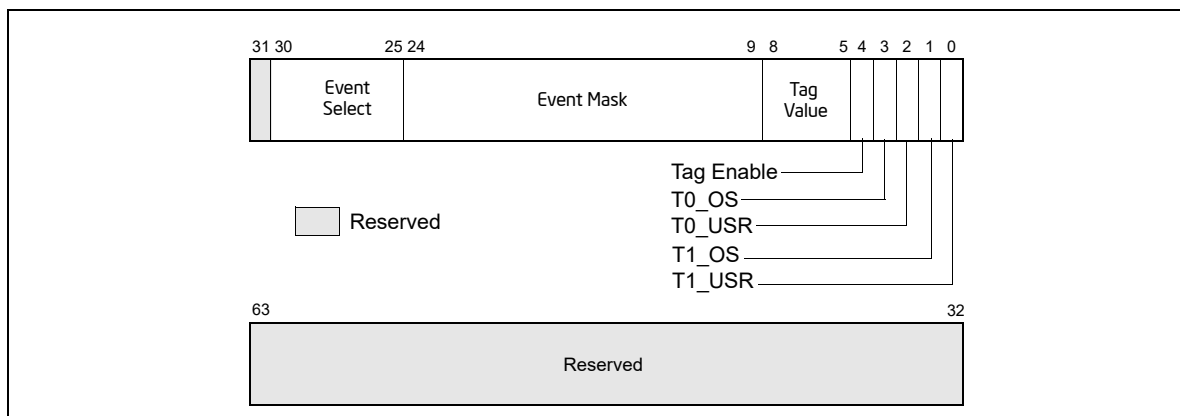


Figure 18-49. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology

- **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 μ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, "At-Retirement Counting."
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a μ op 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 18.6.4.4, "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.

18.6.4.2 CCCR MSRs

Figure 18-50 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.

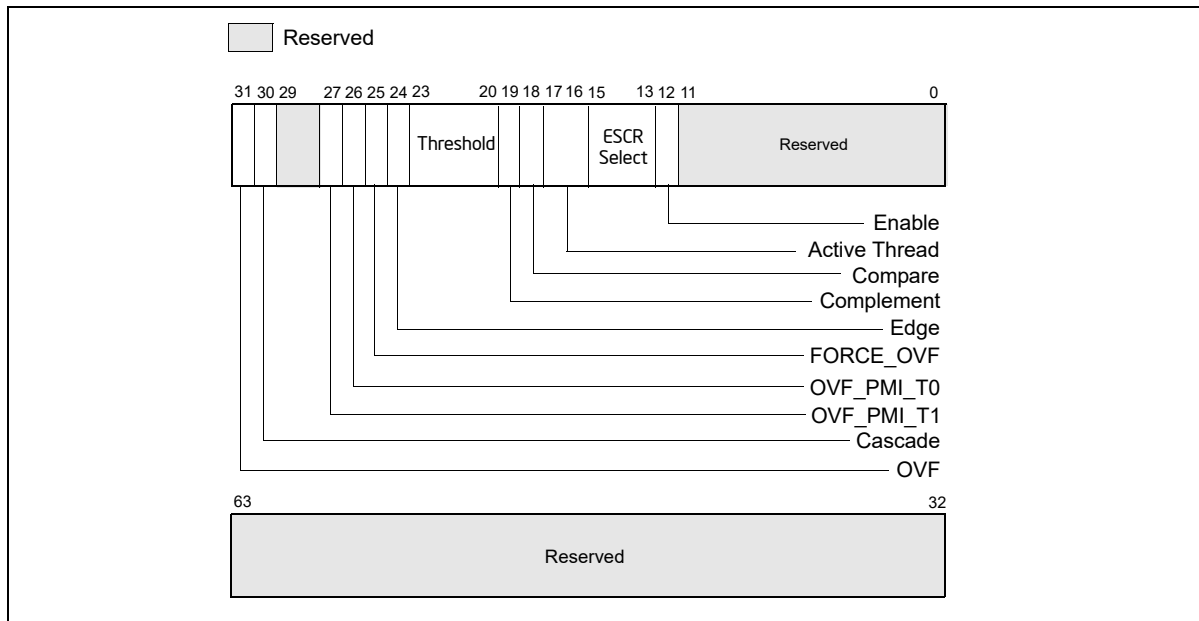


Figure 18-50. Counter Configuration Control Register (CCCR)

- **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 18.6.3.5.2, “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 18.6.3.5.2, “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 18.6.3.2, “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.

18.6.4.3 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.

18.6.4.4 Performance Monitoring Events

All of the events listed in Table 19-33 and 19-34 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 19-39 gives logical processor specific information (TS or TI) for each of the events described in Tables 19-33 and 19-34. If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-83) 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 18-83. 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 18-84.

Table 18-84. 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

18.6.4.5 Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

18.6.4.5.1 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.

18.6.4.5.2 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 processors 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.

18.6.5 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 18.6.4, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst[®] Microarchitecture").

18.6.6 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 18.6.4) 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 18-51.

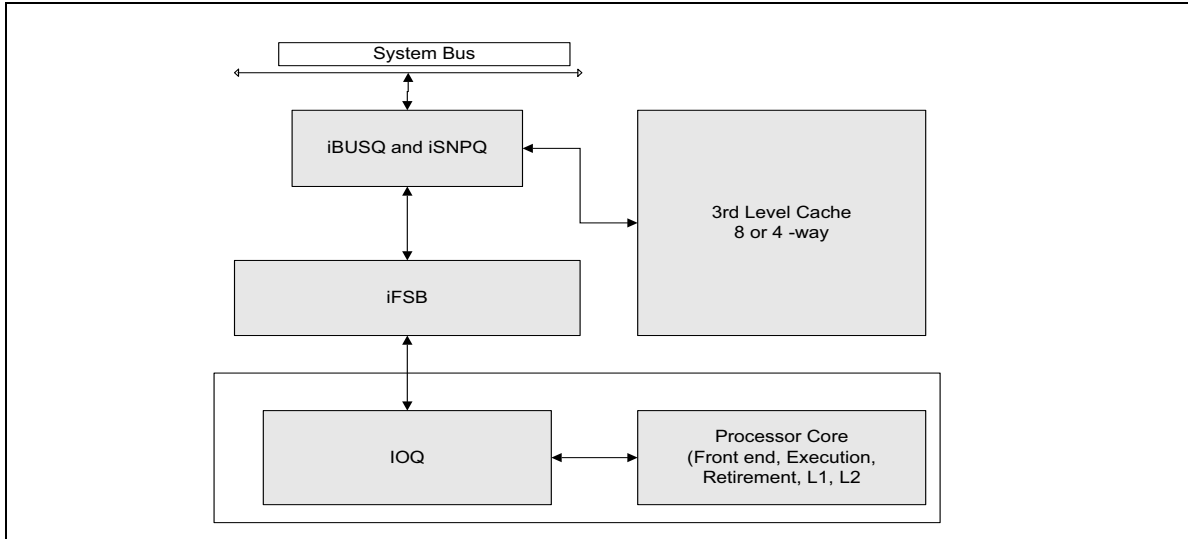


Figure 18-51. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

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 18-52.

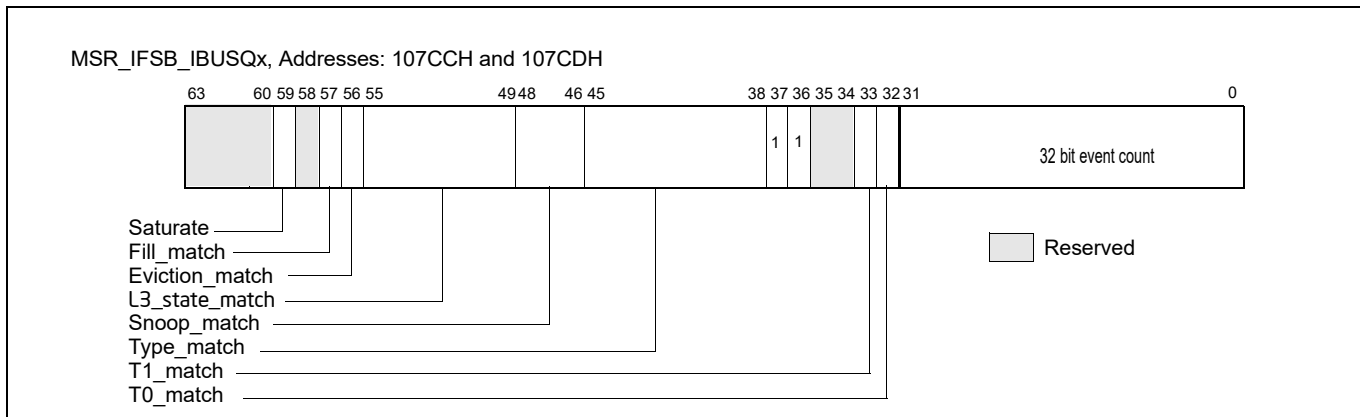


Figure 18-52. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- 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 18-53.

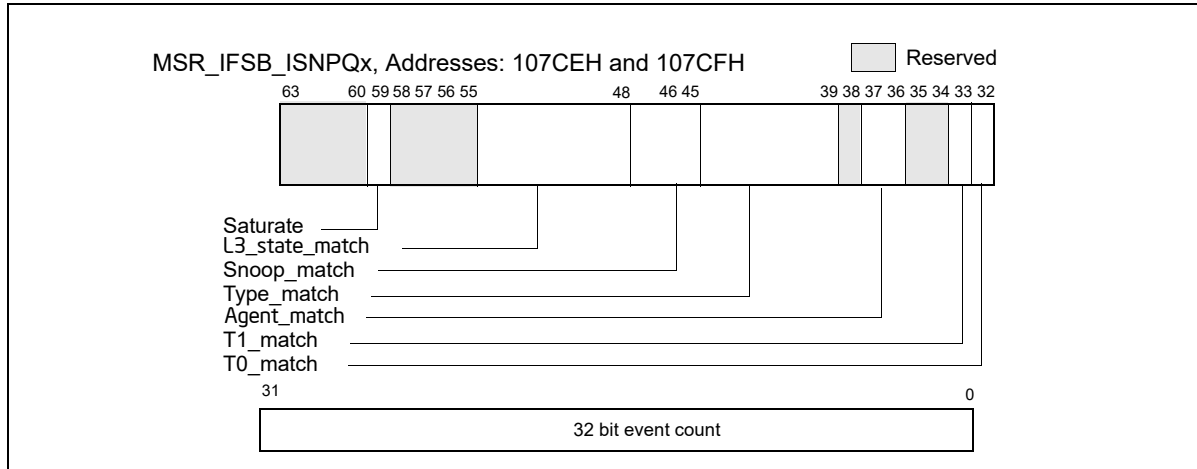


Figure 18-53. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- 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 18-54.

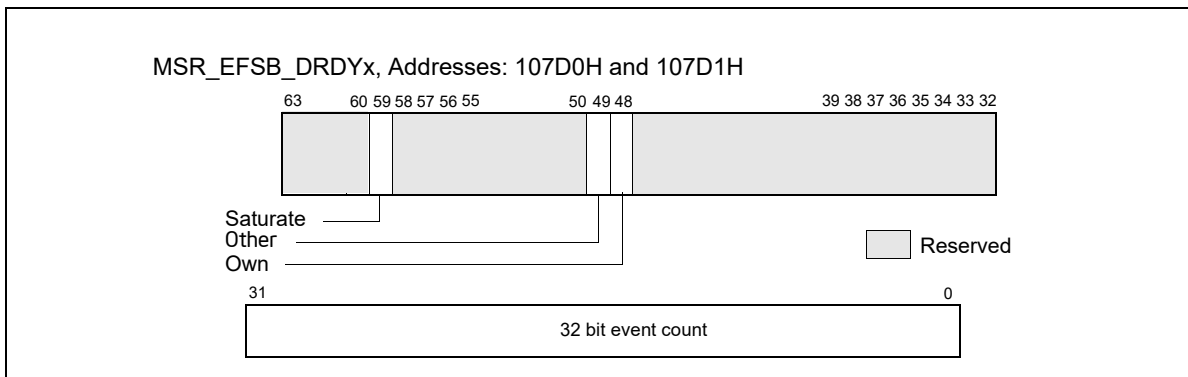
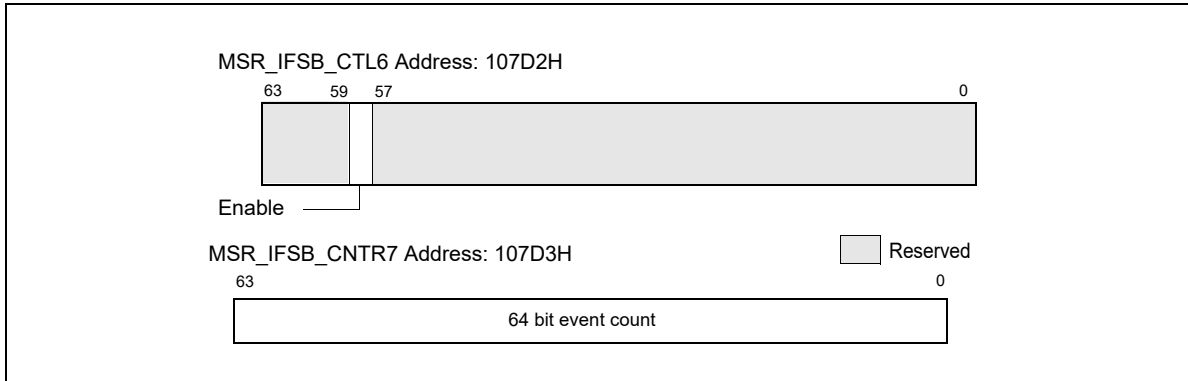


Figure 18-54. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- 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 18-55.



**Figure 18-55. MSR_IFSB_CTL6, Address: 107D2H;
MSR_IFSB_CNTR7, Address: 107D3H**

18.6.7 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 18-56 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-56 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.

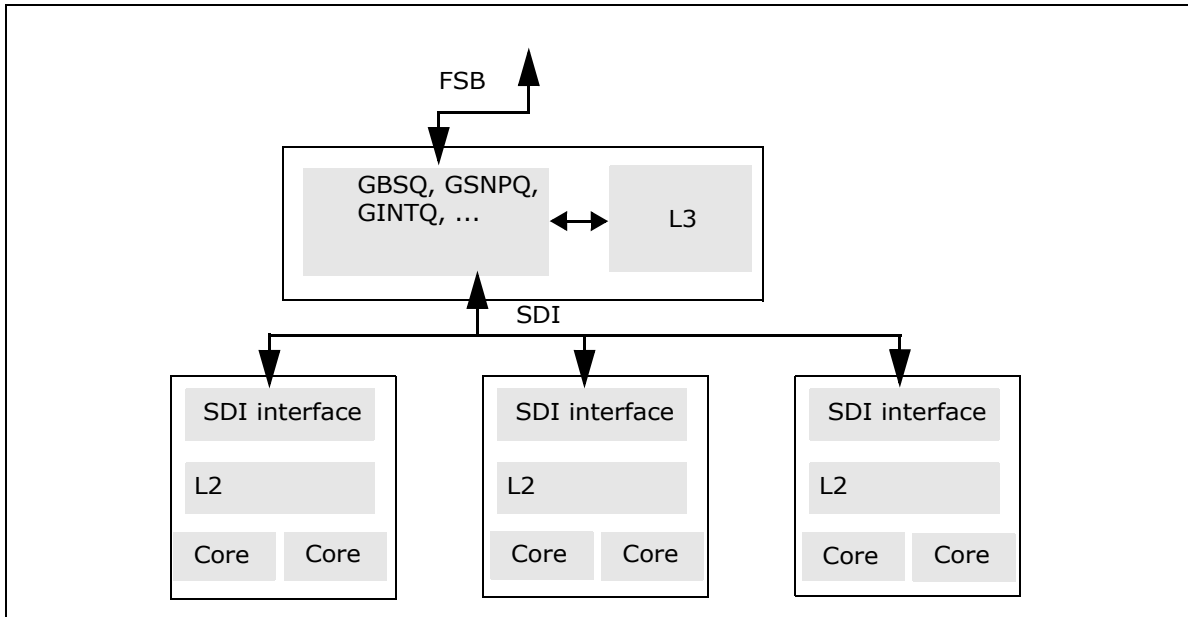


Figure 18-56. Block Diagram of Intel Xeon Processor 7400 Series

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.6.4) apply to Intel Xeon processor 7100 series. The MSR's 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 MSR's 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.

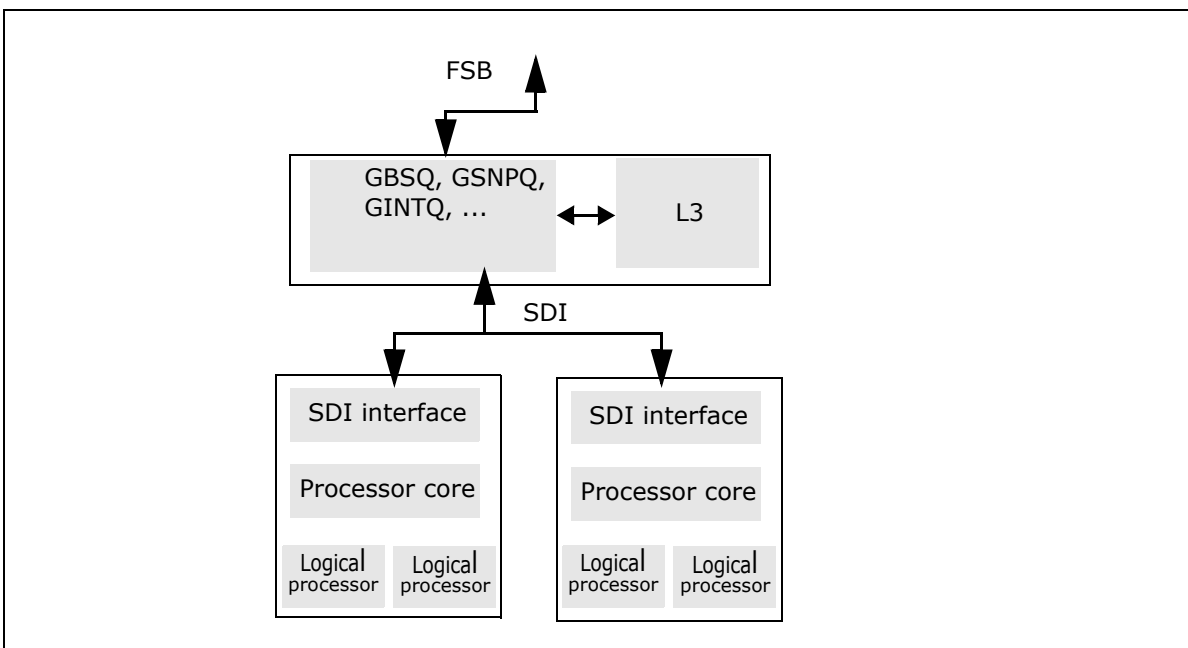


Figure 18-57. Block Diagram of Intel Xeon Processor 7100 Series

18.6.7.1 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.

18.6.7.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 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 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.

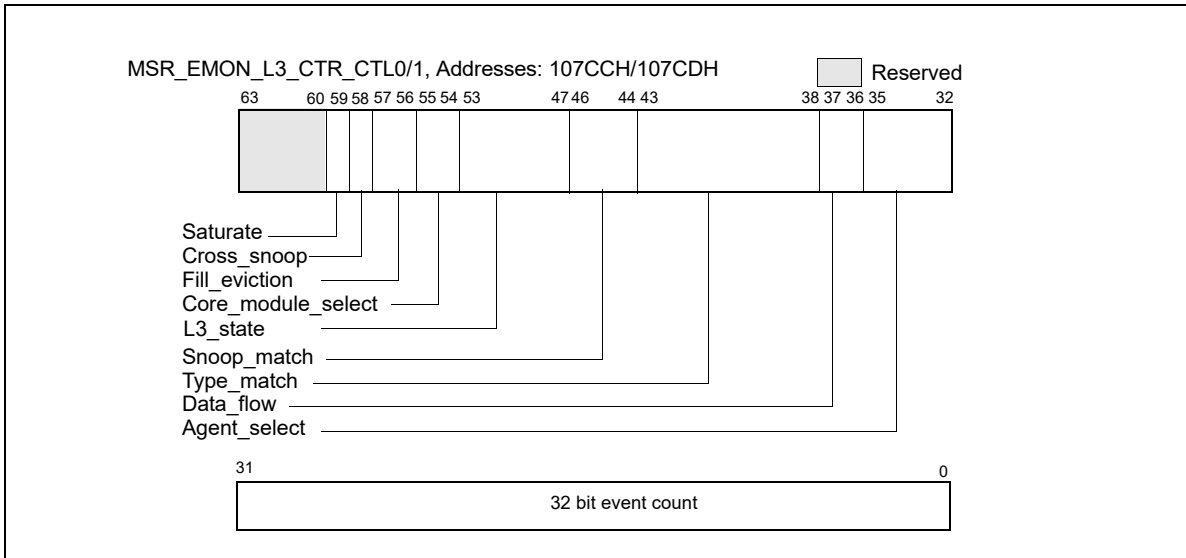


Figure 18-58. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- 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.

18.6.7.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 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) 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.

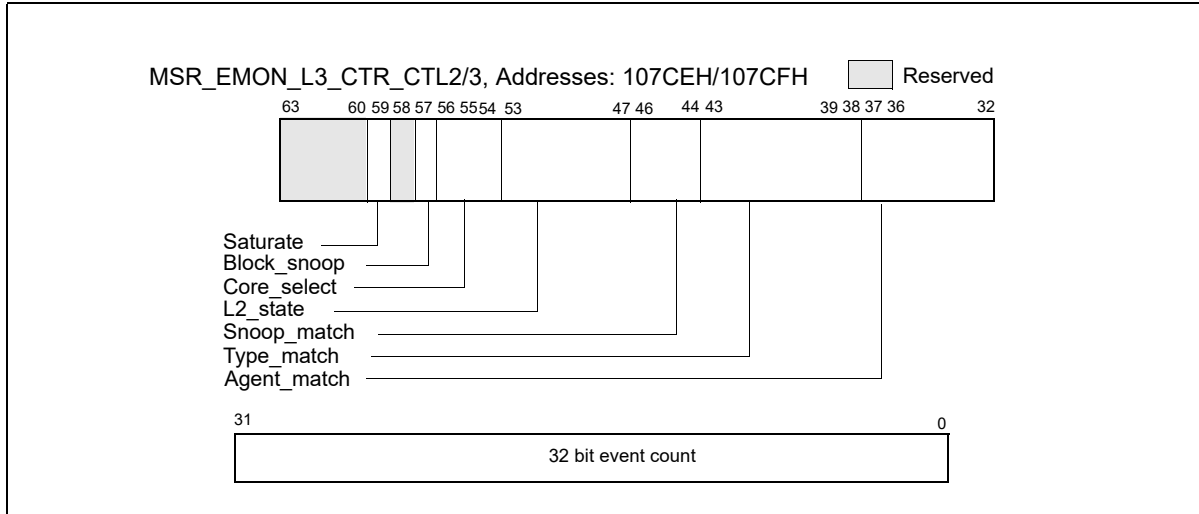


Figure 18-59. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

18.6.7.4 FSB Event Interface

The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 18-60. 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.

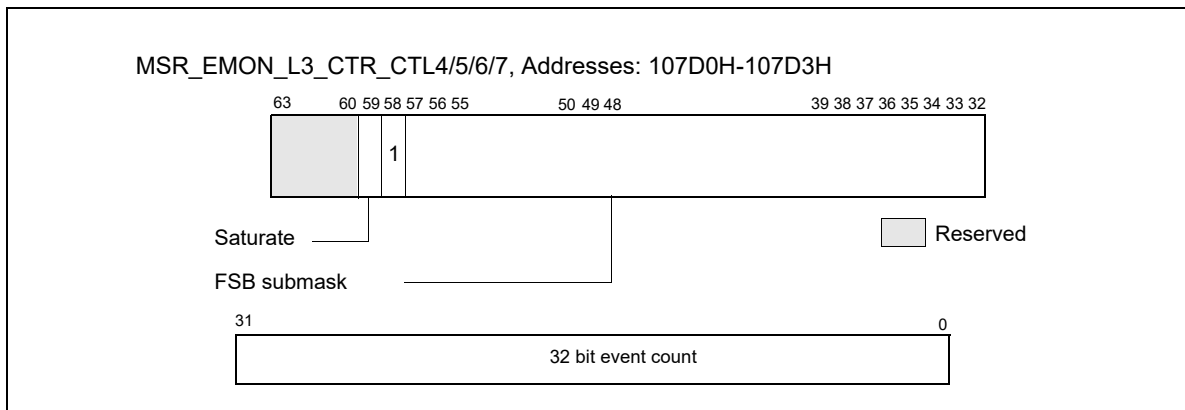


Figure 18-60. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

18.6.7.4.1 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.

18.6.7.5 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 18-58 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

18.6.8 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 19-42, 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 19-42 are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

18.6.8.1 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 18-61 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 19-42, 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 19-42).

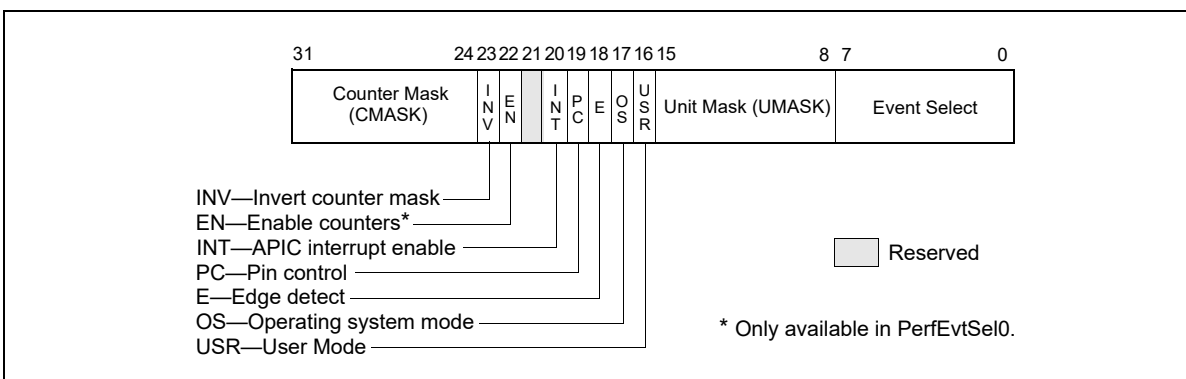


Figure 18-61. PerfEvtSel0 and PerfEvtSel1 MSRs

- **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 PMi pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PMi 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.

18.6.8.2 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.

18.6.8.3 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.

18.6.8.4 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 18.6.8.3, “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.

18.6.8.5 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.

18.6.9 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 19-43 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.

18.6.9.1 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 18-62). 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 19-43 for a list of available event codes.

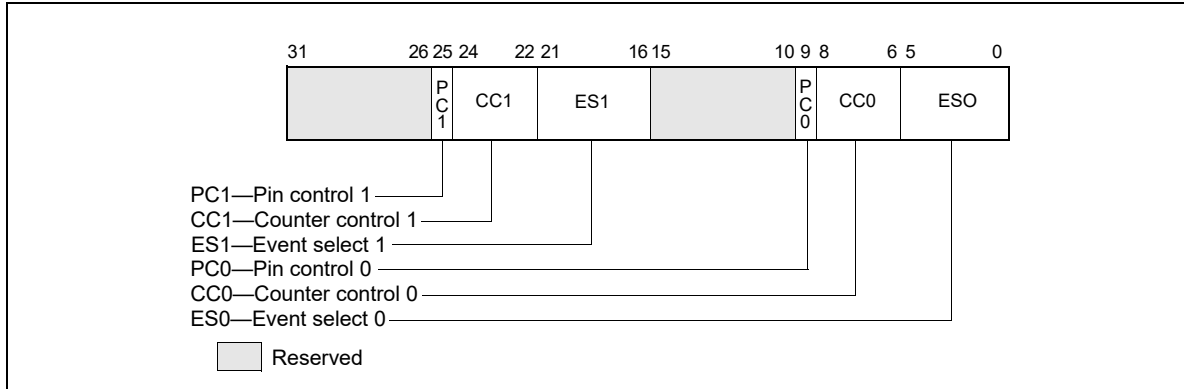


Figure 18-62. CESR MSR (Pentium Processor Only)

- **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 signaling 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.

18.6.9.2 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 than $2^{40} - 1$. 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.

18.6.9.3 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.

18.7 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 18.7.2 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 17.17, “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 17.17, “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.

18.7.1 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.

18.7.2 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.

18.7.3 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:

$$\text{Nominal TSC frequency} = (\text{CPUID.15H.ECX}[31:0] * \text{CPUID.15H.EBX}[31:0]) \div \text{CPUID.15H.EAX}[31:0]$$

For Intel processors in which CPUID.15H.EBX[31:0] \div CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 18-85 can be used to look up the nominal core crystal clock frequency.

Table 18-85. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series ¹	Nominal Core Crystal Clock Frequency
Intel® Xeon® Processor Scalable Family with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.	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.

18.7.3.1 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.

18.7.3.2 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.

18.7.3.3 For Intel® Atom™ Processors Based on the Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

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 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*.

18.7.3.4 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.

18.8 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 18-63; 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 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers; see Section 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records; see Section 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1, see Section 18.8.1.
- 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 18.2.6.
- IA32_PERF_CAPABILITIES.PEBS_BASELINE [bit 14]: If set, the following is true:
 - The IA32_PEBS_ENABLE MSR (address 3F1H) exists and all architecturally enumerated fixed and general-purpose counters have corresponding bits in IA32_PEBS_ENABLE that enable generation of PEBS records. The general-purpose counter bits start at bit IA32_PEBS_ENABLE[0], and the fixed counter bits start at bit IA32_PEBS_ENABLE[32].
 - The format of the PEBS record is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 18.6.2.4.2.
 - Extended PEBS is supported. All counters support the PEBS facility, and all events (both precise and non-precise) can generate PEBS records when PEBS is enabled for that counter. Note that not all events may be available on all counters.
 - Adaptive PEBS is supported. The PEBS_DATA_CFG MSR (address 3F2H) and adaptive record enable bits (IA32_PERFEVTSELx.Adaptive_Record and IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record) are supported. The definition of the PEBS_DATA_CFG MSR, including which bits are supported and how they affect the record, is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 18.9.2.3.
- IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[15]: If set, indicates that the architecture provides built in support for TMA L1 metrics through the IA32_PERF_METRICS MSR, see Section 18.3.9.3.
- IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16]: If set on parts that enumerate support for Intel PT (CPUID.0x7.0.EBX[25]=1), setting IA32_PEBS_ENABLE.PEBS_OUTPUT to 01B will result in PEBS output being written into the Intel PT trace stream. See Section 18.5.5.2.

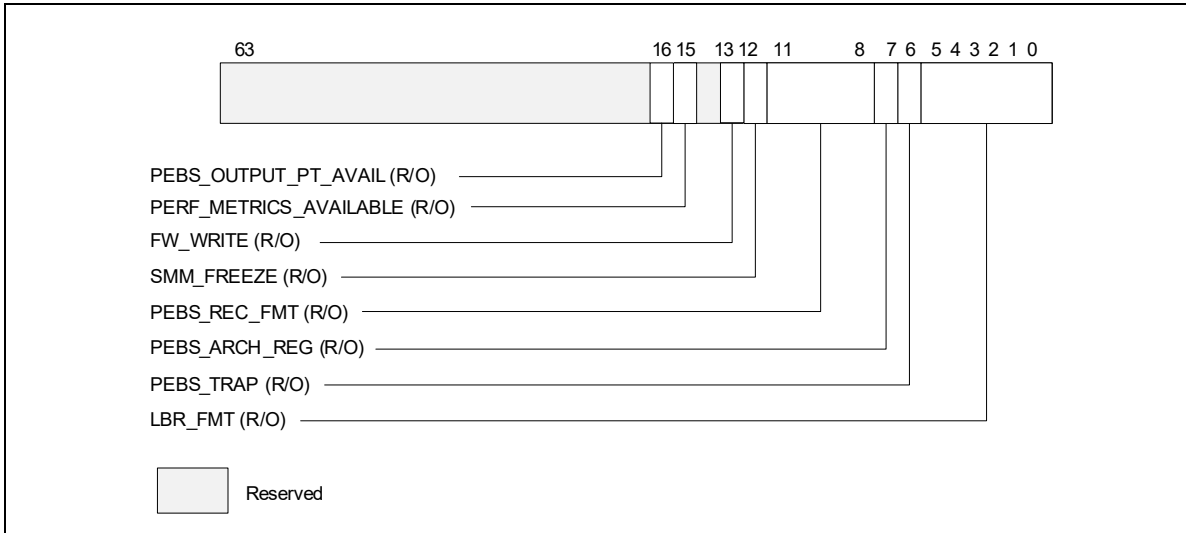


Figure 18-63. Layout of IA32_PERF_CAPABILITIES MSR

18.8.1 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[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

18.9 PEBS FACILITY

18.9.1 Extended PEBS

- The Extended PEBS feature supports Processor Event Based Sampling (PEBS) on all counters, both fixed function and general purpose; and all performance monitoring events, both precise and non-precise. PEBS can be enabled for the general purpose counters using PEBS_EN_PMCi bits of IA32_PEBS_ENABLE (i = 0, 1,...n). PEBS can be enabled for 'i' fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PEBS_ENABLE (i = 0, 1, ...m).

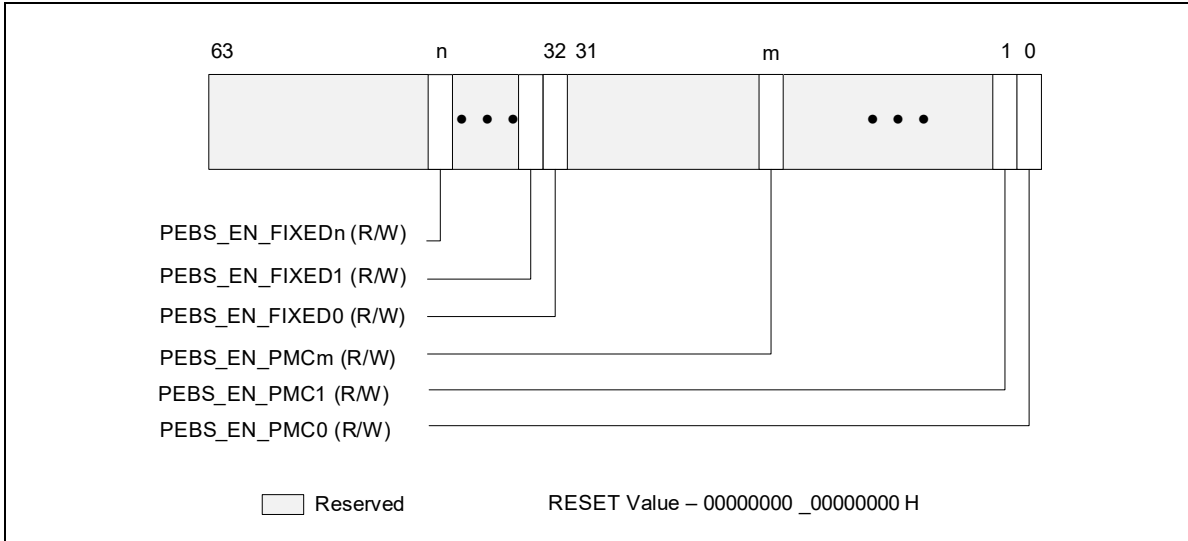


Figure 18-64. Layout of IA32_PEBS_ENABLE MSR

A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

Currently, IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1, IA32_FIXED_CTR2 ... IA32_FIXED_CTRm count as non-precise events.

The Applicable Counter field in the Basic Info Group of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32_PEBS_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

- To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS_BUFFER_MANAGEMENT_AREA data structure that is indicated by the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 18-65. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).

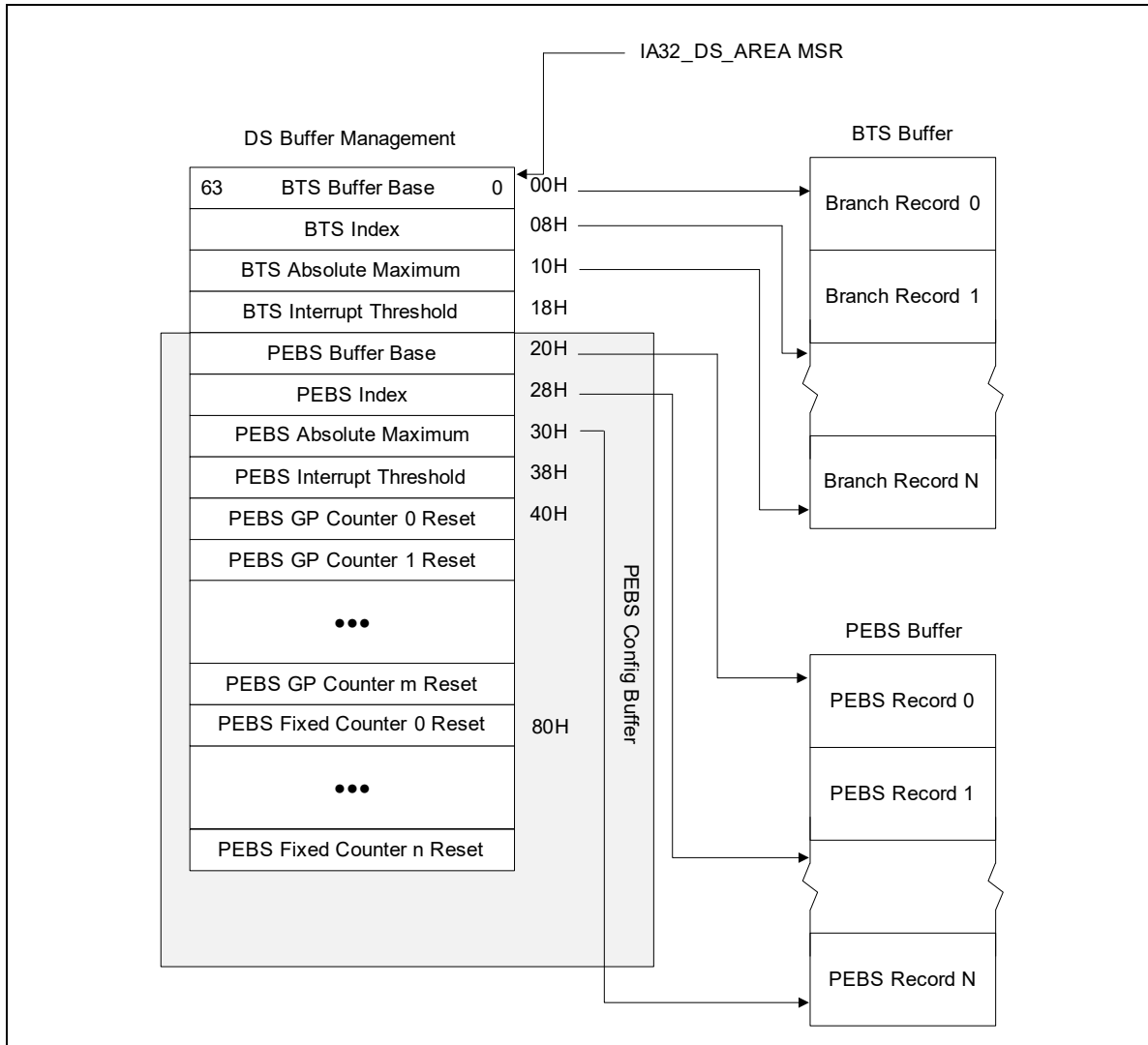


Figure 18-65. PEBS Programming Environment

Extended PEBS support debuts on Intel® Atom processors based on the Goldmont Plus microarchitecture and future Intel® Core™ processors based on the Ice Lake microarchitecture.

18.9.2 Adaptive PEBS

The PEBS facility has been enhanced to collect the following CPU state in addition to GPRs, EventingIP, TSC and memory access related information collected by legacy PEBS:

- XMM registers
- LBR records (TO/FROM/INFO)

The PEBS record is restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group and LBR group. A new register MSR_PEBS_DATA_CFG provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.

By default, the PEBS record will only contain the Basic group. Optionally, each counter can be configured to generate a PEBS records with the groups specified in MSR_PEBS_DATA_CFG.

Details and examples for the Adaptive PEBS capability follow below.

18.9.2.1 Adaptive_Record Counter Control

- IA32_PERFEVTSELx.Adaptive_Record[34]: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_PMCx is set for the corresponding GP counter, an overflow of PMCx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.
- IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_FIXEDx is set for the corresponding Fixed counter, an overflow of FixedCtrx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

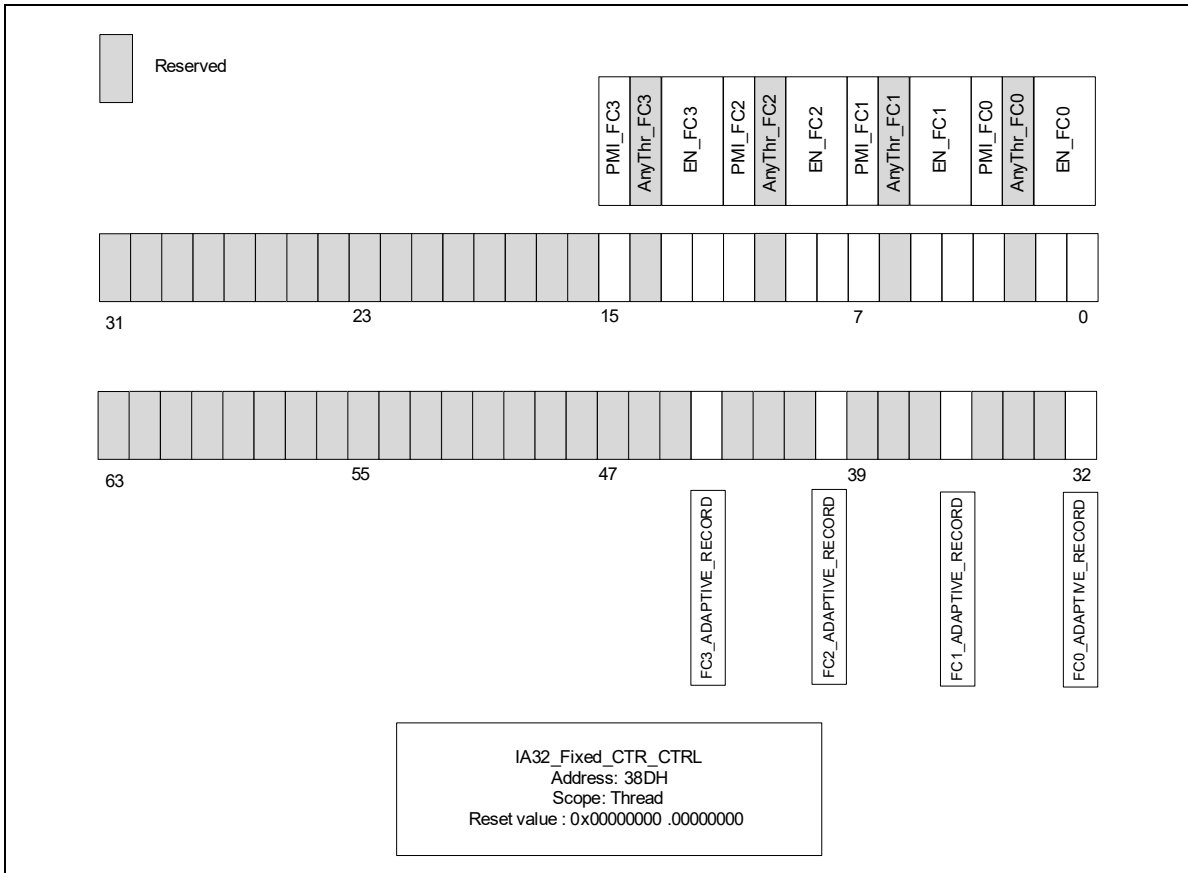


Figure 18-66. Layout of IA32_FIXED_CTR_CTRL MSR Supporting Adaptive PEBS

18.9.2.2 PEBS Record Format

The data fields in the PEBS record are aggregated into five groups which are described in the sub-sections below. Processors that support Adaptive PEBS implement a new MSR called MSR_PEBS_DATA_CFG which allows software to select the data groups to be captured. The data groups are not placed at fixed locations in the PEBS record, but are positioned immediately after one another, thus making the record format/size variable based on the groups selected.

18.9.2.2.1 Basic Info

The Basic group contains essential information for software to parse a record along with several critical fields. It is always collected.

Table 18-86. Basic Info Group

Field Name	Bit Width	Description
Record Format	[47:0]	This field indicates which data groups are included in the record. The field is zero if none of the counters that triggered the current PEBS record have their Adaptive_Record bit set. Otherwise it contains the value of MSR_PEBS_DATA_CFG.
	[63:48]	This field provides the size of the current record in bytes. Selected groups are packed back-to-back in the record without gaps or padding for unselected groups.
Instruction Pointer	[63:0]	This field reports the Eventing Instruction Pointer (EventingIP) of the retired instruction that triggered the PEBS record generation. Note that this field is different than R/EIP which records the instruction pointer of the next instruction to be executed after record generation. The legacy R/EIP field has been removed.
Applicable Counters	[63:0]	The Applicable Counters field indicates which counters triggered the generation of the PEBS record, linking the record to specific events. This allows software to correlate the PEBS record entry properly with the instruction that caused the event, even when multiple counters are configured to generate PEBS records and multiple bits are set in the field.
TSC	[63:0]	This field provides the time stamp counter value when the PEBS record was generated.

18.9.2.2.2 Memory Access Info

This group contains the legacy PEBS memory-related fields; see Section 18.3.1.1.2.

Table 18-87. Memory Access Info Group

Field Name	Bit Width	Description
Memory Access Address	[63:0]	This field contains the linear address of the source of the load, or linear address of the destination (target) of the store. This value is written as a 64-bit address in canonical form.
Memory Auxiliary Info	[63:0]	When MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains an encoded value indicating the memory hierarchy source which satisfied the load. These encodings are detailed in Table 18-4 and Table 18-13. If the PEBS assist was triggered for a store uop, this field will contain information indicating the status of the store, as detailed in Table 18-14.
Memory Access Latency	[63:0]	When MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains the latency to service the load in core clock cycles.
TSX Auxiliary Info	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	[63:31]	This field contains the abort details. Refer to Section 18.3.6.5.1.

18.9.2.2.3 GPRs

This group is captured when the GPR bit is enabled in MSR_PEBS_DATA_CFG. GPRs are always 64 bits wide. If they are selected for non 64-bit mode, the upper 32-bit of the legacy RAX - RDI and all contents of R8-15 GPRs will be filled with 0s. In 64bit mode, the full 64 bit value of each register is written.

The order differs from legacy. The table below shows the order of the GPRs in Ice Lake microarchitecture.

Table 18-88. GPRs in Ice Lake Microarchitecture

Field Name	Bit Width
RFLAGS	[63:0]
RIP	[63:0]
RAX	[63:0]
RCX*	[63:0]
RDX*	[63:0]
RBX*	[63:0]
RSP*	[63:0]
RBP*	[63:0]
RSI*	[63:0]
RDI*	[63:0]
R8	[63:0]
...	...
R15	[63:0]

The machine state reported in the PEBS record is the committed machine state immediately after the instruction that triggers PEBS completes.

For instance, consider the following instruction sequence:

```
MOV eax, [eax]; triggers PEBS record generation
NOP
```

If the mov instruction triggers PEBS record generation, the EventingIP field in the PEBS record will report the address of the mov, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. And the value of RIP will contain the linear address of the nop.

18.9.2.2.4 XMMs

This group is captured when the XMM bit is enabled in MSR_PEBS_DATA_CFG and SSE is enabled. If SSE is not enabled, the fields will contain zeroes. XMM8-XMM15 will also contain zeroes if not in 64-bit mode.

Table 18-89. XMMs

Field Name	Bit Width
XMM0	[127:0]
...	...
XMM15	[127:0]

18.9.2.2.5 LBRs

To capture LBR data in the PEBS record, the LBR bit in MSR_PEBS_DATA_CFG must be enabled. The number of LBR entries included in the record can be configured in the LBR_entries field of MSR_PEBS_DATA_CFG.

Table 18-90. LBRs

Field Name	Bit Width	Description
LBR[<i>i</i>].FROM	[63:0]	Branch from address.
LBR[<i>i</i>].TO	[63:0]	Branch to address.
LBR[<i>i</i>].INFO	[63:0]	Other LBR information, like timing. This field is described in more detail in Section 17.12.1, "MSR_LBR_INFO_x MSR".

LBR entries are recorded into the record starting at LBR[TOS] and proceeding to LBR[TOS-1] and following. Note that LBR index is modulo the number of LBRs supporting on the processor.

18.9.2.3 MSR_PEBS_DATA_CFG

Bits in MSR_PEBS_DATA_CFG can be set to include data field blocks/groups into adaptive records. The Basic Info group is always included in the record. Additionally, the number of LBR entries included in the record is configurable.

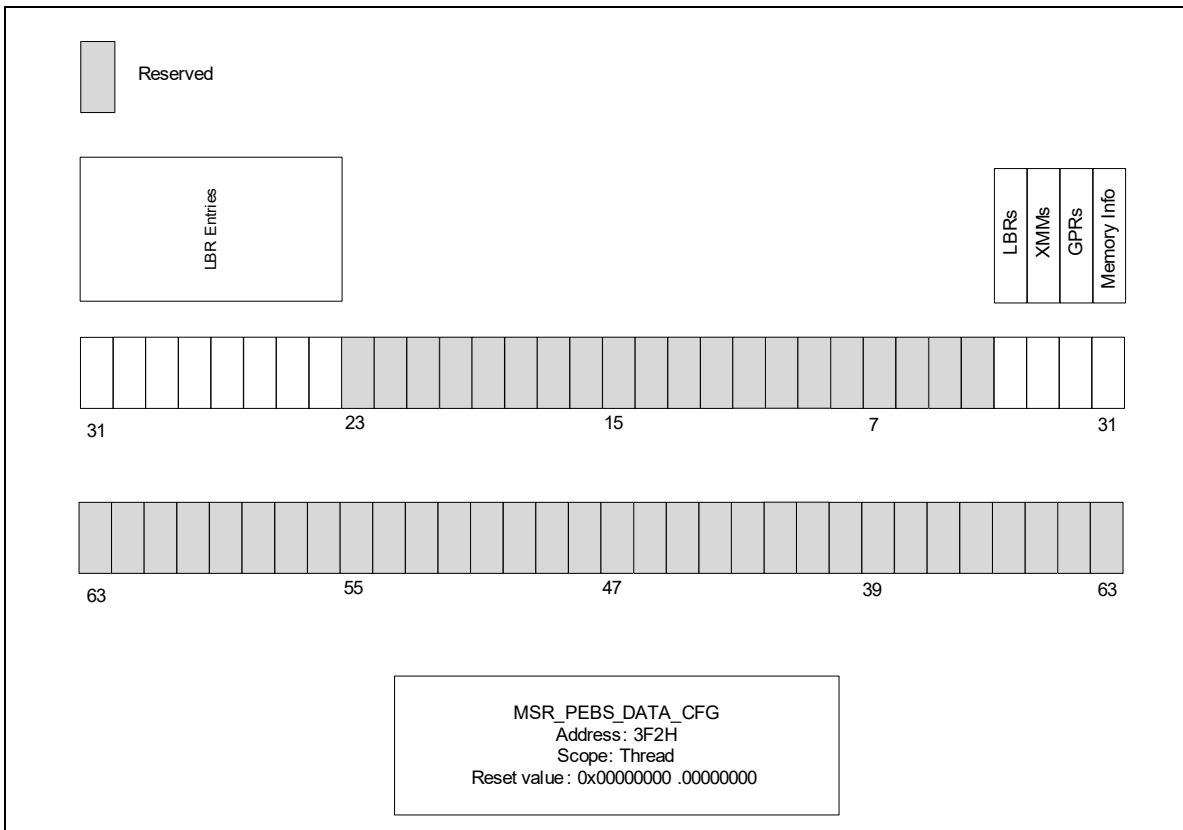


Figure 18-67. MSR_PEBS_DATA_CFG

Table 18-91. MSR_PEBS_CFG Programming¹

Bit	Bit Index	Access	Description
Memory Info	0	R/W	Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.
GPRs	1	R/W	Setting this bit will capture the contents of the General Purpose registers in the PEBS record.
XMMs	2	R/W	Setting this bit will capture the contents of the XMM registers in the PEBS record.
LBRs	3	R/W	Setting this bit will capture LBR TO, FROM and INFO in the PEBS record.
Reserved ²	23:4	NA	Reserved
LBR Entries	31:24	R/W	Set the field to the desired number of entries minus 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, it is recommended to select an even number of LBR entries (programmed into LBR_entries as an odd number).

NOTES:

- 1. A write to the MSR will be ignored when IA32_MISC_ENABLE.PERFMON_AVAILABLE is zero (default).
- 2. Writing to the reserved bits will cause a GP fault.

18.9.2.4 PEBS Record Examples

The following example shows the layout of the PEBS record when all data groups are selected (all valid bits in MSR_PEBS_DATA_CFG are set) and maximum number of LBRs are selected. There are no gaps in the PEBS record when a subset of the groups are selected, thus keeping the layout compact. Implementations that do not support some features will have to pad zeroes in the corresponding fields.

Table 18-92. PEBS Record Example 1

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	Memory Info	Memory Access Address	DLA
0x28		Memory Auxiliary Info	DATA_SRC
0x30		Memory Access Latency	Load Latency
0x38		TSX Auxiliary Info	HLE Information
0x40	GPRs	RFLAGS	
0x48		RIP	
0x50		RAX	
...		...	
0x88		RDI	
0x90		R8	
...		...	
0xC8		R15	
0xD0	XMMs	XMM0	New
...		...	
0x1C0		XMM15	
0x1D0	LBRs	LBR[TOS].FROM	New
0x1D8		LBR[TOS].TO	
0x1E0		LBR[TOS].INFO	
...		...	
0x4B8		LBR[TOS + 1].FROM	
0x4C0		LBR[TOS + 1].TO	
0x4C8		LBR[TOS + 1].INFO	

The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

Table 18-93. PEBS Record Example 2

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	GPRs	RFLAGS	
0x28		RIP	
0x30		RAX	
...		...	
0x68		RDI	
0x70		R8	
...		...	
0xA8		R15	
0xB0	LBRs	LBR[TOS].FROM	New
0xB8		LBR[TOS].TO	
0xC0		LBR[TOS].INFO	
...		...	
0xE0		LBR[TOS + 1].FROM	
0xE8		LBR[TOS + 1].TO	
0xF0		LBR[TOS + 1].INFO	

18.9.3 Precise Distribution of Instructions Retired (PDIR) Facility

Precise Distribution of Instructions Retired Facility is available via PEBS on some microarchitectures. Refer to Section 18.3.4.4.4. Counters that support PDIR also vary. See the processor specific sections for availability.

18.9.4 Reduced Skid PEBS

Processors based on Goldmont Plus microarchitecture support the Reduced Skid PEBS feature described in Section 18.5.3.1.2 on the IA32_PMC0 counter. Although Extended PEBS adds support for generating PEBS records for precise events on additional general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

26. Updates to Chapter 24, Volume 3C

Change bars and green text show changes to Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

Changes to chapter: Added details on protection keys for supervisor-mode pages. Typo corrections and updates as needed.

24.1 OVERVIEW

A logical processor uses **virtual-machine control data structures (VMCSs)** while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor.

A logical processor associates a region in memory with each VMCS. This region is called the **VMCS region**.¹ Software references a specific VMCS using the 64-bit physical address of the region (a **VMCS pointer**). VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). These pointers must not set bits beyond the processor's physical-address width.^{2,3}

A logical processor may maintain a number of VMCSs that are **active**. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. At any given time, at most one of the active VMCSs is the **current** VMCS. (This document frequently uses the term "the VMCS" to refer to the current VMCS.) The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

The following items describe how a logical processor determines which VMCSs are active and which is current:

- The memory operand of the VMPTRLD instruction is the address of a VMCS. After execution of the instruction, that VMCS is both active and current on the logical processor. Any other VMCS that had been active remains so, but no other VMCS is current.
- The VMCS link pointer field in the current VMCS (see Section 24.4.2) is itself the address of a VMCS. If VM entry is performed successfully with the 1-setting of the "VMCS shadowing" VM-execution control, the VMCS referenced by the VMCS link pointer field becomes active on the logical processor. The identity of the current VMCS does not change.
- The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, that VMCS is neither active nor current on the logical processor. If the VMCS had been current on the logical processor, the logical processor no longer has a current VMCS.

The VMPTRST instruction stores the address of the logical processor's current VMCS into a specified memory location (it stores the value FFFFFFFF_FFFFFFFFH if there is no current VMCS).

The **launch state** of a VMCS determines which VM-entry instruction should be used with that VMCS: the VMLAUNCH instruction requires a VMCS whose launch state is "clear"; the VMRESUME instruction requires a VMCS whose launch state is "launched". A logical processor maintains a VMCS's launch state in the corresponding VMCS region. The following items describe how a logical processor manages the launch state of a VMCS:

- If the launch state of the current VMCS is "clear", successful execution of the VMLAUNCH instruction changes the launch state to "launched".
- The memory operand of the VMCLEAR instruction is the address of a VMCS. After execution of the instruction, the launch state of that VMCS is "clear".
- There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to discover it (it cannot be read using VMREAD).

1. The amount of memory required for a VMCS region is at most 4 KBytes. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).

2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

3. If IA32_VMX_BASIC[48] is read as 1, these pointers must not set any bits in the range 63:32; see Appendix A.1.

Figure 24-1 illustrates the different states of a VMCS. It uses "X" to refer to the VMCS and "Y" to refer to any other VMCS. Thus: "VMPTRLD X" always makes X current and active; "VMPTRLD Y" always makes X not current (because it makes Y current); "VMLAUNCH" makes the launch state of X "launched" if X was current and its launch state was "clear"; and "VMCLEAR X" always makes X inactive and not current and makes its launch state "clear".

The figure does not illustrate operations that do not modify the VMCS state relative to these parameters (e.g., execution of VMPTRLD X when X is already current). Note that VMCLEAR X makes X "inactive, not current, and clear," even if X's current state is not defined (e.g., even if X has not yet been initialized). See Section 24.11.3.

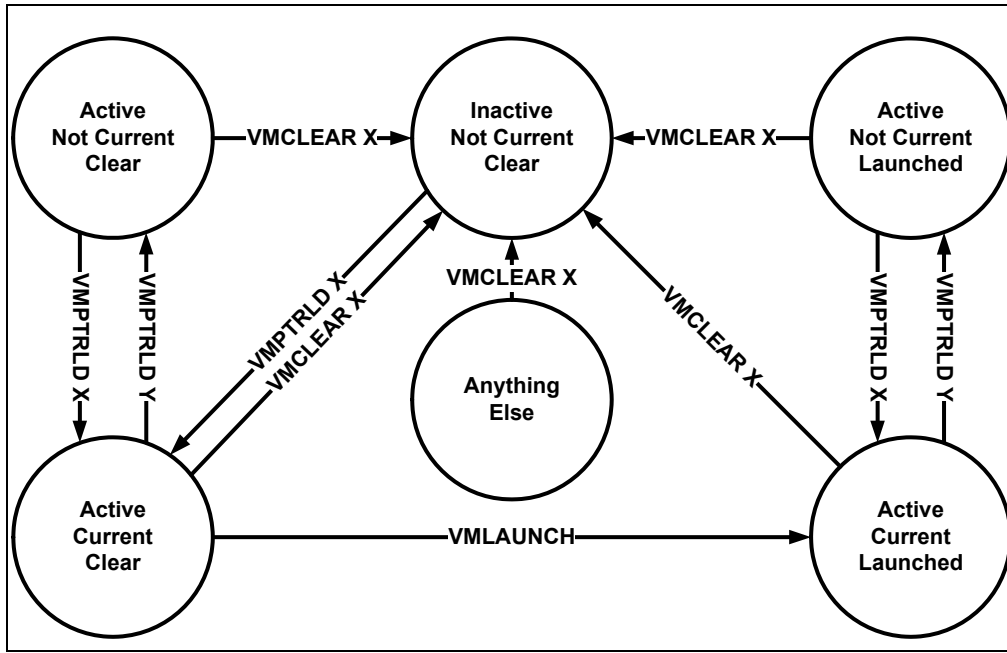


Figure 24-1. States of VMCS X

Because a shadow VMCS (see Section 24.10) cannot be used for VM entry, the launch state of a shadow VMCS is not meaningful. Figure 24-1 does not illustrate all the ways in which a shadow VMCS may be made active.

24.2 FORMAT OF THE VMCS REGION

A VMCS region comprises up to 4-KBytes.¹ The format of a VMCS region is given in Table 24-1.

Table 24-1. Format of the VMCS Region

Byte Offset	Contents
0	Bits 30:0: VMCS revision identifier Bit 31: shadow-VMCS indicator (see Section 24.10)
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

1. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).

The first 4 bytes of the VMCS region contain the **VMCS revision identifier** at bits 30:0.¹ Processors that maintain VMCS data in different formats (see below) use different VMCS revision identifiers. These identifiers enable software to avoid using a VMCS region formatted for one processor on a processor that uses a different format.² Bit 31 of this 4-byte region indicates whether the VMCS is a shadow VMCS (see Section 24.10).

Software should write the VMCS revision identifier to the VMCS region before using that region for a VMCS. The VMCS revision identifier is never written by the processor; VMPTRLD fails if its operand references a VMCS region whose VMCS revision identifier differs from that used by the processor. (VMPTRLD also fails if the shadow-VMCS indicator is 1 and the processor does not support the 1-setting of the “VMCS shadowing” VM-execution control; see Section 24.6.2) Software can discover the VMCS revision identifier that a processor uses by reading the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

Software should clear or set the shadow-VMCS indicator depending on whether the VMCS is to be an ordinary VMCS or a shadow VMCS (see Section 24.10). VMPTRLD fails if the shadow-VMCS indicator is set and the processor does not support the 1-setting of the “VMCS shadowing” VM-execution control. Software can discover support for this setting by reading the VMX capability MSR IA32_VMX_PROCBASED_CTLS2 (see Appendix A.3.3).

The next 4 bytes of the VMCS region are used for the **VMX-abort indicator**. The contents of these bits do not control processor operation in any way. A logical processor writes a non-zero value into these bits if a VMX abort occurs (see Section 27.7). Software may also write into this field.

The remainder of the VMCS region is used for **VMCS data** (those parts of the VMCS that control VMX non-root operation and the VMX transitions). The format of these data is implementation-specific. VMCS data are discussed in Section 24.3 through Section 24.9. To ensure proper behavior in VMX operation, software should maintain the VMCS region and related structures (enumerated in Section 24.11.4) in writeback cacheable memory. Future implementations may allow or require a different memory type³. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

24.3 ORGANIZATION OF VMCS DATA

The VMCS data are organized into six logical groups:

- **Guest-state area.** Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
- **Host-state area.** Processor state is loaded from the host-state area on VM exits.
- **VM-execution control fields.** These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits.
- **VM-exit control fields.** These fields control VM exits.
- **VM-entry control fields.** These fields control VM entries.
- **VM-exit information fields.** These fields receive information on VM exits and describe the cause and the nature of VM exits. On some processors, these fields are read-only.⁴

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are sometimes referred to collectively as VMX controls.

-
1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.
 2. Logical processors that use the same VMCS revision identifier use the same size for VMCS regions.
 3. Alternatively, software may map any of these regions or structures with the UC memory type. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32_VMX_BASIC with exceptions noted in Appendix A.1.
 4. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

24.4 GUEST-STATE AREA

This section describes fields contained in the guest-state area of the VMCS. VM entries load processor state from these fields and VM exits store processor state into these fields. See Section 26.3.2 and Section 27.3 for details.

24.4.1 Guest Register State

The following fields in the guest-state area correspond to processor registers:

- Control registers CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Debug register DR7 (64 bits; 32 bits on processors that do not support Intel 64 architecture).
- RSP, RIP, and RFLAGS (64 bits each; 32 bits on processors that do not support Intel 64 architecture).¹
- The following fields for each of the registers CS, SS, DS, ES, FS, GS, LDTR, and TR:
 - Selector (16 bits).
 - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture). The base-address fields for CS, SS, DS, and ES have only 32 architecturally-defined bits; nevertheless, the corresponding VMCS fields have 64 bits on processors that support Intel 64 architecture.
 - Segment limit (32 bits). The limit field is always a measure in bytes.
 - Access rights (32 bits). The format of this field is given in Table 24-2 and detailed as follows:
 - The low 16 bits correspond to bits 23:8 of the upper 32 bits of a 64-bit segment descriptor. While bits 19:16 of code-segment and data-segment descriptors correspond to the upper 4 bits of the segment limit, the corresponding bits (bits 11:8) are reserved in this VMCS field.
 - Bit 16 indicates an **unusable segment**. Attempts to use such a segment fault except in 64-bit mode. In general, a segment register is unusable if it has been loaded with a null selector.²
 - Bits 31:17 are reserved.

Table 24-2. Format of Access Rights

Bit Position(s)	Field
3:0	Segment type
4	S — Descriptor type (0 = system; 1 = code or data)
6:5	DPL — Descriptor privilege level
7	P — Segment present
11:8	Reserved
12	AVL — Available for use by system software

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
2. There are a few exceptions to this statement. For example, a segment with a non-null selector may be unusable following a task switch that fails after its commit point; see “Interrupt 10—Invalid TSS Exception (#TS)” in Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. In contrast, the TR register is usable after processor reset despite having a null selector; see Table 10-1 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Table 24-2. Format of Access Rights (Contd.)

Bit Position(s)	Field
13	Reserved (except for CS) L — 64-bit mode active (for CS only)
14	D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
15	G — Granularity
16	Segment unusable (0 = usable; 1 = unusable)
31:17	Reserved

The base address, segment limit, and access rights compose the “hidden” part (or “descriptor cache”) of each segment register. These data are included in the VMCS because it is possible for a segment register’s descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register’s selector.

The value of the DPL field for SS is always equal to the logical processor’s current privilege level (CPL).¹

On some processors, executions of VMWRITE ignore attempts to write non-zero values to any of bits 11:8 or bits 31:17. On such processors, VMREAD always returns 0 for those bits, and VM entry treats those bits as if they were all 0 (see Section 26.3.1.2).

- The following fields for each of the registers GDTR and IDTR:
 - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture).
 - Limit (32 bits). The limit fields contain 32 bits even though these fields are specified as only 16 bits in the architecture.
- The following MSRs:
 - IA32_DEBUGCTL (64 bits)
 - IA32_SYSENTER_CS (32 bits)
 - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture)
 - IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32_PERF_GLOBAL_CTRL” VM-entry control.
 - IA32_PAT (64 bits). This field is supported only on processors that support either the 1-setting of the “load IA32_PAT” VM-entry control or that of the “save IA32_PAT” VM-exit control.
 - IA32_EFER (64 bits). This field is supported only on processors that support either the 1-setting of the “load IA32_EFER” VM-entry control or that of the “save IA32_EFER” VM-exit control.
 - IA32_BNDCFGS (64 bits). This field is supported only on processors that support either the 1-setting of the “load IA32_BNDCFGS” VM-entry control or that of the “clear IA32_BNDCFGS” VM-exit control.
 - IA32_RTIT_CTL (64 bits). This field is supported only on processors that support either the 1-setting of the “load IA32_RTIT_CTL” VM-entry control or that of the “clear IA32_RTIT_CTL” VM-exit control.
 - IA32_S_CET (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.
 - IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.
 - IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the “load PKRS” VM-entry control.

1. In protected mode, CPL is also associated with the RPL field in the CS selector. However, the RPL fields are not meaningful in real-address mode or in virtual-8086 mode.

- The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.
- The register SMBASE (32 bits). This register contains the base address of the logical processor’s SMRAM image.

24.4.2 Guest Non-Register State

In addition to the register state described in Section 24.4.1, the guest-state area includes the following fields that characterize guest state but which do not correspond to processor registers:

- **Activity state** (32 bits). This field identifies the logical processor’s activity state. When a logical processor is executing instructions normally, it is in the **active state**. Execution of certain instructions and the occurrence of certain events may cause a logical processor to transition to an **inactive state** in which it ceases to execute instructions.

The following activity states are defined:¹

- 0: **Active**. The logical processor is executing instructions normally.
- 1: **HLT**. The logical processor is inactive because it executed the HLT instruction.
- 2: **Shutdown**. The logical processor is inactive because it incurred a **triple fault**² or some other serious error.
- 3: **Wait-for-SIPI**. The logical processor is inactive because it is waiting for a startup-IPI (SIPI).

Future processors may include support for other activity states. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine what activity states are supported.

- **Interruptibility state** (32 bits). The IA-32 architecture includes features that permit certain events to be blocked for a period of time. This field contains information about such blocking. Details and the format of this field are given in Table 24-3.

Table 24-3. Format of Interruptibility State

Bit Position(s)	Bit Name	Notes
0	Blocking by STI	See the “STI—Set Interrupt Flag” section in Chapter 4 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B</i> . Execution of STI with RFLAGS.IF = 0 blocks maskable interrupts on the instruction boundary following its execution. ¹ Setting this bit indicates that this blocking is in effect.
1	Blocking by MOV SS	See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> . Execution of a MOV to SS or a POP to SS blocks or suppresses certain debug exceptions as well as interrupts (maskable and nonmaskable) on the instruction boundary following its execution. Setting this bit indicates that this blocking is in effect. ² This document uses the term “blocking by MOV SS,” but it applies equally to POP SS.
2	Blocking by SMI	See Section 34.2, “System Management Interrupt (SMI).” System-management interrupts (SMIs) are disabled while the processor is in system-management mode (SMM). Setting this bit indicates that blocking of SMIs is in effect.

1. Execution of the MWAIT instruction may put a logical processor into an inactive state. However, this VMCS field never reflects this state. See Section 27.1.

2. A triple fault occurs when a logical processor encounters an exception while attempting to deliver a double fault.

Table 24-3. Format of Interruptibility State (Contd.)

Bit Position(s)	Bit Name	Notes
3	Blocking by NMI	See Section 6.7.1, “Handling Multiple NMIs,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> and Section 34.8, “NMI Handling While in SMM.” Delivery of a non-maskable interrupt (NMI) or a system-management interrupt (SMI) blocks subsequent NMIs until the next execution of IRET. See Section 25.3 for how this behavior of IRET may change in VMX non-root operation. Setting this bit indicates that blocking of NMIs is in effect. Clearing this bit does not imply that NMIs are not (temporarily) blocked for other reasons. If the “virtual NMIs” VM-execution control (see Section 24.6.1) is 1, this bit does not control the blocking of NMIs. Instead, it refers to “virtual-NMI blocking” (the fact that guest software is not ready for an NMI).
4	Enclave interruption	Set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode. A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.
31:5	Reserved	VM entry will fail if these bits are not 0. See Section 26.3.1.5.

NOTES:

1. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.
 2. System-management interrupts may also be inhibited on the instruction boundary following such an execution of MOV or POP.
- **Pending debug exceptions** (64 bits; 32 bits on processors that do not support Intel 64 architecture). IA-32 processors may recognize one or more debug exceptions without immediately delivering them.¹ This field contains information about such exceptions. This field is described in Table 24-4.

Table 24-4. Format of Pending-Debug-Exceptions

Bit Position(s)	Bit Name	Notes
3:0	B3 - B0	When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if the corresponding enabling bit in DR7 is not set.
11:4	Reserved	VM entry fails if these bits are not 0. See Section 26.3.1.5.
12	Enabled breakpoint	When set, this bit indicates that at least one data or I/O breakpoint was met and was enabled in DR7.
13	Reserved	VM entry fails if this bit is not 0. See Section 26.3.1.5.
14	BS	When set, this bit indicates that a debug exception would have been triggered by single-step execution mode.
15	Reserved	VM entry fails if this bit is not 0. See Section 26.3.1.5.

1. For example, execution of a MOV to SS or a POP to SS may inhibit some debug exceptions for one instruction. See Section 6.8.3 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. In addition, certain events incident to an instruction (for example, an INIT signal) may take priority over debug traps generated by that instruction. See Table 6-2 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Table 24-4. Format of Pending-Debug-Exceptions (Contd.)

Bit Position(s)	Bit Name	Notes
16	RTM	When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i>). ¹
63:17	Reserved	VM entry fails if these bits are not 0. See Section 26.3.1.5. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 16 to indicate an RTM-related exception, while this field **sets** the bit to indicate that condition.

- **VMCS link pointer** (64 bits). If the "VMCS shadowing" VM-execution control is 1, the VMREAD and VMWRITE instructions access the VMCS referenced by this pointer (see Section 24.10). Otherwise, software should set this field to FFFFFFFF_FFFFFFFFH to avoid VM-entry failures (see Section 26.3.1.5).
- **VMX-preemption timer value** (32 bits). This field is supported only on processors that support the 1-setting of the "activate VMX-preemption timer" VM-execution control. This field contains the value that the VMX-preemption timer will use following the next VM entry with that setting. See Section 25.5.1 and Section 26.7.4.
- **Page-directory-pointer-table entries** (PDPTEs; 64 bits each). These four (4) fields (PDPTE0, PDPTE1, PDPTE2, and PDPTE3) are supported only on processors that support the 1-setting of the "enable EPT" VM-execution control. They correspond to the PDPTEs referenced by CR3 when PAE paging is in use (see Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). They are used only if the "enable EPT" VM-execution control is 1.
- **Guest interrupt status** (16 bits). This field is supported only on processors that support the 1-setting of the "virtual-interrupt delivery" VM-execution control. It characterizes part of the guest's virtual-APIC state and does not correspond to any processor or APIC registers. It comprises two 8-bit subfields:
 - **Requesting virtual interrupt (RVI)**. This is the low byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is requesting service. (The value 0 implies that there is no such interrupt.)
 - **Servicing virtual interrupt (SVI)**. This is the high byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is in service. (The value 0 implies that there is no such interrupt.)

See Chapter 29 for more information on the use of this field.
- **PML index** (16 bits). This field is supported only on processors that support the 1-setting of the "enable PML" VM-execution control. It contains the logical index of the next entry in the page-modification log. Because the page-modification log comprises 512 entries, the PML index is typically a value in the range 0–511. Details of the page-modification log and use of the PML index are given in Section 28.2.6.

24.5 HOST-STATE AREA

This section describes fields contained in the host-state area of the VMCS. As noted earlier, processor state is loaded from these fields on every VM exit (see Section 27.5).

All fields in the host-state area correspond to processor registers:

- CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- RSP and RIP (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Selector fields (16 bits each) for the segment registers CS, SS, DS, ES, FS, GS, and TR. There is no field in the host-state area for the LDTR selector.

- Base-address fields for FS, GS, TR, GDTR, and IDTR (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- The following MSRs:
 - IA32_SYSENTER_CS (32 bits)
 - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture).
 - IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32_PERF_GLOBAL_CTRL” VM-exit control.
 - IA32_PAT (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32_PAT” VM-exit control.
 - IA32_EFER (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32_EFER” VM-exit control.
 - IA32_S_CET (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.
 - IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.
 - IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the “load PKRS” VM-exit control.
- The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.

In addition to the state identified here, some processor state components are loaded with fixed values on every VM exit; there are no fields corresponding to these components in the host-state area. See Section 27.5 for details of how state is loaded on VM exits.

24.6 VM-EXECUTION CONTROL FIELDS

The VM-execution control fields govern VMX non-root operation. These are described in Section 24.6.1 through Section 24.6.8.

24.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts).¹ Table 24-5 lists the controls. See Chapter 27 for how these controls affect processor behavior in VMX non-root operation.

1. Some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls (see Section 25.2).

Table 24-5. Definitions of Pin-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	External-interrupt exiting	If this control is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.
3	NMI exiting	If this control is 1, non-maskable interrupts (NMIs) cause VM exits. Otherwise, they are delivered normally using descriptor 2 of the IDT. This control also determines interactions between IRET and blocking by NMI (see Section 25.3).
5	Virtual NMIs	If this control is 1, NMIs are never blocked and the “blocking by NMI” bit (bit 3) in the interruptibility-state field indicates “virtual-NMI blocking” (see Table 24-3). This control also interacts with the “NMI-window exiting” VM-execution control (see Section 24.6.2).
6	Activate VMX-preemption timer	If this control is 1, the VMX-preemption timer counts down in VMX non-root operation; see Section 25.5.1. A VM exit occurs when the timer counts down to zero; see Section 25.2.
7	Process posted interrupts	If this control is 1, the processor treats interrupts with the posted-interrupt notification vector (see Section 24.6.8) specially, updating the virtual-APIC page with posted-interrupt requests (see Section 29.6).

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PINBASED_CTLs and IA32_VMX_TRUE_PINBASED_CTLs (see Appendix A.3.1) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 26.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 2, and 4. The VMX capability MSR IA32_VMX_PINBASED_CTLs will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PINBASED_CTLs MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

24.6.2 Processor-Based VM-Execution Controls

The processor-based VM-execution controls constitute two 32-bit vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions.¹ These are the **primary processor-based VM-execution controls** and the **secondary processor-based VM-execution controls**.

Table 24-6 lists the primary processor-based VM-execution controls. See Chapter 25 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 24.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 24.6.5 and Section 25.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPIC exiting	This control determines whether executions of RDPIC cause VM exits.
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.

1. Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls (see Section 25.1.2), as do task switches (see Section 25.2).

Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
15	CR3-load exiting	In conjunction with the CR3-target controls (see Section 24.6.7), this control determines whether executions of MOV to CR3 cause VM exits. See Section 25.1.3. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
16	CR3-store exiting	This control determines whether executions of MOV from CR3 cause VM exits. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
19	CR8-load exiting	This control determines whether executions of MOV to CR8 cause VM exits.
20	CR8-store exiting	This control determines whether executions of MOV from CR8 cause VM exits.
21	Use TPR shadow	Setting this control to 1 enables TPR virtualization and other APIC-virtualization features. See Chapter 29.
22	NMI-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if there is no virtual-NMI blocking (see Section 24.4.2).
23	MOV-DR exiting	This control determines whether executions of MOV DR cause VM exits.
24	Unconditional I/O exiting	This control determines whether executions of I/O instructions (IN, INS/INSB/INSD, OUT, and OUTS/OUTSB/OUTSW/OUTSD) cause VM exits.
25	Use I/O bitmaps	This control determines whether I/O bitmaps are used to restrict executions of I/O instructions (see Section 24.6.4 and Section 25.1.3). For this control, "0" means "do not use I/O bitmaps" and "1" means "use I/O bitmaps." If the I/O bitmaps are used, the setting of the "unconditional I/O exiting" control is ignored.
27	Monitor trap flag	If this control is 1, the monitor trap flag debugging feature is enabled. See Section 25.5.2.
28	Use MSR bitmaps	This control determines whether MSR bitmaps are used to control execution of the RDMSR and WRMSR instructions (see Section 24.6.9 and Section 25.1.3). For this control, "0" means "do not use MSR bitmaps" and "1" means "use MSR bitmaps." If the MSR bitmaps are not used, all executions of the RDMSR and WRMSR instructions cause VM exits.
29	MONITOR exiting	This control determines whether executions of MONITOR cause VM exits.
30	PAUSE exiting	This control determines whether executions of PAUSE cause VM exits.
31	Activate secondary controls	This control determines whether the secondary processor-based VM-execution controls are used. If this control is 0, the logical processor operates as if all the secondary processor-based VM-execution controls were also 0.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PROCBASED_CTLs and IA32_VMX_TRUE_PROCBASED_CTLs (see Appendix A.3.2) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 26.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 4–6, 8, 13–16, and 26. The VMX capability MSR IA32_VMX_PROCBASED_CTLs will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PROCBASED_CTLs MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

Bit 31 of the primary processor-based VM-execution controls determines whether the secondary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the secondary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 31 of the primary processor-based VM-execution controls do not support the secondary processor-based VM-execution controls.

Table 24-7 lists the secondary processor-based VM-execution controls. See Chapter 25 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 24-7. Definitions of Secondary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	Virtualize APIC accesses	If this control is 1, the logical processor treats specially accesses to the page with the APIC-access address. See Section 29.4.
1	Enable EPT	If this control is 1, extended page tables (EPT) are enabled. See Section 28.2.
2	Descriptor-table exiting	This control determines whether executions of LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR cause VM exits.
3	Enable RDTSCP	If this control is 0, any execution of RDTSCP causes an invalid-opcode exception (#UD).
4	Virtualize x2APIC mode	If this control is 1, the logical processor treats specially RDMSR and WRMSR to APIC MSRs (in the range 800H-8FFH). See Section 29.5.
5	Enable VPID	If this control is 1, cached translations of linear addresses are associated with a virtual-processor identifier (VPID). See Section 28.1.
6	WBINVD exiting	This control determines whether executions of WBINVD cause VM exits.
7	Unrestricted guest	This control determines whether guest software may run in unpagged protected mode or in real-address mode.
8	APIC-register virtualization	If this control is 1, the logical processor virtualizes certain APIC accesses. See Section 29.4 and Section 29.5.
9	Virtual-interrupt delivery	This controls enables the evaluation and delivery of pending virtual interrupts as well as the emulation of writes to the APIC registers that control interrupt prioritization.
10	PAUSE-loop exiting	This control determines whether a series of executions of PAUSE can cause a VM exit (see Section 24.6.13 and Section 25.1.3).
11	RDRAND exiting	This control determines whether executions of RDRAND cause VM exits.
12	Enable INVPCID	If this control is 0, any execution of INVPCID causes a #UD.
13	Enable VM functions	Setting this control to 1 enables use of the VMFUNC instruction in VMX non-root operation. See Section 25.5.6.
14	VMCS shadowing	If this control is 1, executions of VMREAD and VMWRITE in VMX non-root operation may access a shadow VMCS (instead of causing VM exits). See Section 24.10 and Section 30.3.
15	Enable ENCLS exiting	If this control is 1, executions of ENCLS consult the ENCLS-exiting bitmap to determine whether the instruction causes a VM exit. See Section 24.6.16 and Section 25.1.3.
16	RDSEED exiting	This control determines whether executions of RDSEED cause VM exits.
17	Enable PML	If this control is 1, an access to a guest-physical address that sets an EPT dirty bit first adds an entry to the page-modification log. See Section 28.2.6.
18	EPT-violation #VE	If this control is 1, EPT violations may cause virtualization exceptions (#VE) instead of VM exits. See Section 25.5.7.
19	Conceal VMX from PT	If this control is 1, Intel Processor Trace suppresses from PIPs an indication that the processor was in VMX non-root operation and omits a VMCS packet from any PSB+ produced in VMX non-root operation (see Chapter 35).
20	Enable XSAVES/XRSTORS	If this control is 0, any execution of XSAVES or XRSTORS causes a #UD.
22	Mode-based execute control for EPT	If this control is 1, EPT execute permissions are based on whether the linear address being accessed is supervisor mode or user mode. See Chapter 28.
23	Sub-page write permissions for EPT	If this control is 1, EPT write permissions may be specified at the granularity of 128 bytes. See Section 28.2.4.
24	Intel PT uses guest physical addresses	If this control is 1, all output addresses used by Intel Processor Trace are treated as guest-physical addresses and translated using EPT. See Section 25.5.4.

Table 24-7. Definitions of Secondary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
25	Use TSC scaling	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC multiplier field (see Section 24.6.5 and Section 25.3).
26	Enable user wait and pause	If this control is 0, any execution of TPAUSE, UMONITOR, or UMWAIT causes a #UD.
28	Enable ENCLV exiting	If this control is 1, executions of ENCLV consult the ENCLV-exiting bitmap to determine whether the instruction causes a VM exit. See Section 24.6.17 and Section 25.1.3.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROCBASED_CTL2 (see Appendix A.3.3) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 26.2.1.1).

24.6.3 Exception Bitmap

The **exception bitmap** is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exception's vector.

Whether a page fault (exception with vector 14) causes a VM exit is determined by bit 14 in the exception bitmap as well as the error code produced by the page fault and two 32-bit fields in the VMCS (the **page-fault error-code mask** and **page-fault error-code match**). See Section 25.2 for details.

24.6.4 I/O-Bitmap Addresses

The VM-execution control fields include the 64-bit physical addresses of **I/O bitmaps** A and B (each of which are 4 KBytes in size). I/O bitmap A contains one bit for each I/O port in the range 0000H through 7FFFH; I/O bitmap B contains bits for ports in the range 8000H through FFFFH.

A logical processor uses these bitmaps if and only if the "use I/O bitmaps" control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if any bit in the I/O bitmaps corresponding to a port it accesses is 1. See Section 25.1.3 for details. If the bitmaps are used, their addresses must be 4-KByte aligned.

24.6.5 Time-Stamp Counter Offset and Multiplier

The VM-execution control fields include a 64-bit **TSC-offset** field. If the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1, this field controls executions of the RDTSC and RDTSCP instructions. It also controls executions of the RDMSR instruction that read from the IA32_TIME_STAMP_COUNTER MSR. For all of these, the value of the TSC offset is added to the value of the time-stamp counter, and the sum is returned to guest software in EDX:EAX.

Processors that support the 1-setting of the "use TSC scaling" control also support a 64-bit **TSC-multiplier** field. If this control is 1 (and the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1), this field also affects the executions of the RDTSC, RDTSCP, and RDMSR instructions identified above. Specifically, the contents of the time-stamp counter is first multiplied by the TSC multiplier before adding the TSC offset.

See Chapter 25 for a detailed treatment of the behavior of RDTSC, RDTSCP, and RDMSR in VMX non-root operation.

24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4

VM-execution control fields include **guest/host masks** and **read shadows** for the CR0 and CR4 registers. These fields control executions of instructions that access those registers (including CLTS, LMSW, MOV CR, and SMSW). They are 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

In general, bits set to 1 in a guest/host mask correspond to bits "owned" by the host:

- Guest attempts to set them (using CLTS, LMSW, or MOV to CR) to values differing from the corresponding bits in the corresponding read shadow cause VM exits.
- Guest reads (using MOV from CR or SMSW) return values for these bits from the corresponding read shadow.

Bits cleared to 0 correspond to bits “owned” by the guest; guest attempts to modify them succeed and guest reads return values for these bits from the control register itself.

See Chapter 27 for details regarding how these fields affect VMX non-root operation.

24.6.7 CR3-Target Controls

The VM-execution control fields include a set of 4 **CR3-target values** and a **CR3-target count**. The CR3-target values each have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not. The CR3-target count has 32 bits on all processors.

An execution of MOV to CR3 in VMX non-root operation does not cause a VM exit if its source operand matches one of these values. If the CR3-target count is n , only the first n CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit

There are no limitations on the values that can be written for the CR3-target values. VM entry fails (see Section 26.2) if the CR3-target count is greater than 4.

Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine the number of values supported.

24.6.8 Controls for APIC Virtualization

There are three mechanisms by which software accesses registers of the logical processor’s local APIC:

- If the local APIC is in xAPIC mode, it can perform memory-mapped accesses to addresses in the 4-KByte page referenced by the physical address in the IA32_APIC_BASE MSR (see Section 10.4.4, “Local APIC Status and Location” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* and *Intel® 64 Architecture Processor Topology Enumeration*).¹
- If the local APIC is in x2APIC mode, it can access the local APIC’s registers using the RDMSR and WRMSR instructions (see *Intel® 64 Architecture Processor Topology Enumeration*).
- In 64-bit mode, it can access the local APIC’s task-priority register (TPR) using the MOV CR8 instruction.

There are five processor-based VM-execution controls (see Section 24.6.2) that control such accesses. There are “use TPR shadow”, “virtualize APIC accesses”, “virtualize x2APIC mode”, “virtual-interrupt delivery”, and “APIC-register virtualization”. These controls interact with the following fields:

- **APIC-access address** (64 bits). This field contains the physical address of the 4-KByte **APIC-access page**. If the “virtualize APIC accesses” VM-execution control is 1, access to this page may cause VM exits or be virtualized by the processor. See Section 29.4.

The APIC-access address exists only on processors that support the 1-setting of the “virtualize APIC accesses” VM-execution control.

- **Virtual-APIC address** (64 bits). This field contains the physical address of the 4-KByte **virtual-APIC page**. The processor uses the virtual-APIC page to virtualize certain accesses to APIC registers and to manage virtual interrupts; see Chapter 29.

Depending on the setting of the controls indicated earlier, the virtual-APIC page may be accessed by the following operations:

- The MOV CR8 instructions (see Section 29.3).
- Accesses to the APIC-access page if, in addition, the “virtualize APIC accesses” VM-execution control is 1 (see Section 29.4).
- The RDMSR and WRMSR instructions if, in addition, the value of ECX is in the range 800H–8FFH (indicating an APIC MSR) and the “virtualize x2APIC mode” VM-execution control is 1 (see Section 29.5).

1. If the local APIC does not support x2APIC mode, it is always in xAPIC mode.

If the “use TPR shadow” VM-execution control is 1, VM entry ensures that the virtual-APIC address is 4-KByte aligned. The virtual-APIC address exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.

- **TPR threshold** (32 bits). Bits 3:0 of this field determine the threshold below which bits 7:4 of VTPR (see Section 29.1.1) cannot fall. If the “virtual-interrupt delivery” VM-execution control is 0, a VM exit occurs after an operation (e.g., an execution of MOV to CR8) that reduces the value of those bits below the TPR threshold. See Section 29.1.2.

The TPR threshold exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.

- **EOI-exit bitmap** (4 fields; 64 bits each). These fields are supported only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control. They are used to determine which virtualized writes to the APIC’s EOI register cause VM exits:
 - EOI_EXIT0 contains bits for vectors from 0 (bit 0) to 63 (bit 63).
 - EOI_EXIT1 contains bits for vectors from 64 (bit 0) to 127 (bit 63).
 - EOI_EXIT2 contains bits for vectors from 128 (bit 0) to 191 (bit 63).
 - EOI_EXIT3 contains bits for vectors from 192 (bit 0) to 255 (bit 63).

See Section 29.1.4 for more information on the use of this field.

- **Posted-interrupt notification vector** (16 bits). This field is supported only on processors that support the 1-setting of the “process posted interrupts” VM-execution control. Its low 8 bits contain the interrupt vector that is used to notify a logical processor that virtual interrupts have been posted. See Section 29.6 for more information on the use of this field.
- **Posted-interrupt descriptor address** (64 bits). This field is supported only on processors that support the 1-setting of the “process posted interrupts” VM-execution control. It is the physical address of a 64-byte aligned posted interrupt descriptor. See Section 29.6 for more information on the use of this field.

24.6.9 MSR-Bitmap Address

On processors that support the 1-setting of the “use MSR bitmaps” VM-execution control, the VM-execution control fields include the 64-bit physical address of four contiguous **MSR bitmaps**, which are each 1-KByte in size. This field does not exist on processors that do not support the 1-setting of that control. The four bitmaps are:

- **Read bitmap for low MSRs** (located at the MSR-bitmap address). This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Read bitmap for high MSRs** (located at the MSR-bitmap address plus 1024). This contains one bit for each MSR address in the range C0000000H to C0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Write bitmap for low MSRs** (located at the MSR-bitmap address plus 2048). This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.
- **Write bitmap for high MSRs** (located at the MSR-bitmap address plus 3072). This contains one bit for each MSR address in the range C0000000H to C0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

A logical processor uses these bitmaps if and only if the “use MSR bitmaps” control is 1. If the bitmaps are used, an execution of RDMSR or WRMSR causes a VM exit if the value of RCX is in neither of the ranges covered by the bitmaps or if the appropriate bit in the MSR bitmaps (corresponding to the instruction and the RCX value) is 1. See Section 25.1.3 for details. If the bitmaps are used, their address must be 4-KByte aligned.

24.6.10 Executive-VMCS Pointer

The executive-VMCS pointer is a 64-bit field used in the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). SMM VM exits save this field as described in Section 34.15.2. VM entries that return from SMM use this field as described in Section 34.15.4.

24.6.11 Extended-Page-Table Pointer (EPTP)

The **extended-page-table pointer** (EPTP) contains the address of the base of EPT PML4 table (see Section 28.2.2), as well as other EPT configuration information. The format of this field is shown in Table 24-8.

Table 24-8. Format of Extended-Page-Table Pointer

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 28.2.7): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved. ¹
5:3	This value is 1 less than the EPT page-walk length (see Section 28.2.2)
6	Setting this control to 1 enables accessed and dirty flags for EPT (see Section 28.2.5) ²
7	Setting this control to 1 enables enforcement of access rights for supervisor shadow-stack pages (see Section 28.2.3.2) ³
11:8	Reserved
N-1:12	Bits N-1:12 of the physical address of the 4-KByte aligned EPT PML4 table ⁴
63:N	Reserved

NOTES:

1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine what EPT paging-structure memory types are supported.
2. Not all processors support accessed and dirty flags for EPT. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.
3. Not all processors enforce access rights for shadow-stack pages. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.
4. N is the physical-address width supported by the logical processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The EPTP exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.

24.6.12 Virtual-Processor Identifier (VPID)

The **virtual-processor identifier** (VPID) is a 16-bit field. It exists only on processors that support the 1-setting of the “enable VPID” VM-execution control. See Section 28.1 for details regarding the use of this field.

24.6.13 Controls for PAUSE-Loop Exiting

On processors that support the 1-setting of the “PAUSE-loop exiting” VM-execution control, the VM-execution control fields include the following 32-bit fields:

- **PLE_Gap.** Software can configure this field as an upper bound on the amount of time between two successive executions of PAUSE in a loop.
- **PLE_Window.** Software can configure this field as an upper bound on the amount of time a guest is allowed to execute in a PAUSE loop.

These fields measure time based on a counter that runs at the same rate as the timestamp counter (TSC). See Section 25.1.3 for more details regarding PAUSE-loop exiting.

24.6.14 VM-Function Controls

The **VM-function controls** constitute a 64-bit vector that governs use of the VMFUNC instruction in VMX non-root operation. This field is supported only on processors that support the 1-settings of both the “activate secondary controls” primary processor-based VM-execution control and the “enable VM functions” secondary processor-based VM-execution control.

Table 24-9 lists the VM-function controls. See Section 25.5.6 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 24-9. Definitions of VM-Function Controls

Bit Position(s)	Name	Description
0	EPTP switching	The EPTP-switching VM function changes the EPT pointer to a value chosen from the EPTP list. See Section 25.5.6.3.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_VMFUNC (see Appendix A.11) to determine which bits are reserved. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 26.2.1.1).

Processors that support the 1-setting of the “EPTP switching” VM-function control also support a 64-bit field called the **EPTP-list address**. This field contains the physical address of the 4-KByte **EPTP list**. The EPTP list comprises 512 8-Byte entries (each an EPTP value) and is used by the EPTP-switching VM function (see Section 25.5.6.3).

24.6.15 VMCS Shadowing Bitmap Addresses

On processors that support the 1-setting of the “VMCS shadowing” VM-execution control, the VM-execution control fields include the 64-bit physical addresses of the **VMREAD bitmap** and the **VMWRITE bitmap**. Each bitmap is 4 KBytes in size and thus contains 32 KBits. The addresses are the **VMREAD-bitmap address** and the **VMWRITE-bitmap address**.

If the “VMCS shadowing” VM-execution control is 1, executions of VMREAD and VMWRITE may consult these bitmaps (see Section 24.10 and Section 30.3).

24.6.16 ENCLS-Exiting Bitmap

The **ENCLS-exiting bitmap** is a 64-bit field. If the “enable ENCLS exiting” VM-execution control is 1, execution of ENCLS causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 25.1.3 for more information.

24.6.17 ENCLV-Exiting Bitmap

The **ENCLV-exiting bitmap** is a 64-bit field. If the “enable ENCLV exiting” VM-execution control is 1, execution of ENCLV causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 25.1.3 for more information.

24.6.18 Control Field for Page-Modification Logging

The **PML address** is a 64-bit field. It is the 4-KByte aligned address of the **page-modification log**. The page-modification log consists of 512 64-bit entries. It is used for the page-modification logging feature. Details of the page-modification logging are given in Section 28.2.6.

If the “enable PML” VM-execution control is 1, VM entry ensures that the PML address is 4-KByte aligned. The PML address exists only on processors that support the 1-setting of the “enable PML” VM-execution control.

24.6.19 Controls for Virtualization Exceptions

On processors that support the 1-setting of the “EPT-violation #VE” VM-execution control, the VM-execution control fields include the following:

- **Virtualization-exception information address** (64 bits). This field contains the physical address of the **virtualization-exception information area**. When a logical processor encounters a virtualization exception, it saves virtualization-exception information at the virtualization-exception information address; see Section 25.5.7.2.
- **EPTP index** (16 bits). When an EPT violation causes a virtualization exception, the processor writes the value of this field to the virtualization-exception information area. The EPTP-switching VM function updates this field (see Section 25.5.6.3).

24.6.20 XSS-Exiting Bitmap

On processors that support the 1-setting of the “enable XSAVES/XRSTORS” VM-execution control, the VM-execution control fields include a 64-bit **XSS-exiting bitmap**. If the “enable XSAVES/XRSTORS” VM-execution control is 1, executions of XSAVES and XRSTORS may consult this bitmap (see Section 25.1.3 and Section 25.3).

24.6.21 Sub-Page-Permission-Table Pointer (SPPTP)

If the sub-page write-permission feature of EPT is enabled, EPT write permissions may be determined at a 128-byte granularity (see Section 28.2.4). These permissions are determined using a hierarchy of sub-page-permission structures in memory.

The root of this hierarchy is referenced by a VM-execution control field called the **sub-page-permission-table pointer** (SPPTP). The SPPTP contains the address of the base of the root SPP table (see Section 28.2.4.2). The format of this field is shown in Table 24-8.

Table 24-10. Format of Sub-Page-Permission-Table Pointer

Bit Position(s)	Field
11:0	Reserved
N-1:12	Bits N-1:12 of the physical address of the 4-KByte aligned root SPP table
63:N ¹	Reserved

NOTES:

1. N is the processor’s physical-address width. Software can determine this width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The SPPTP exists only on processors that support the 1-setting of the “sub-page write permissions for EPT” VM-execution control.

24.7 VM-EXIT CONTROL FIELDS

The VM-exit control fields govern the behavior of VM exits. They are discussed in Section 24.7.1 and Section 24.7.2.

24.7.1 VM-Exit Controls

The **VM-exit controls** constitute a 32-bit vector that governs the basic operation of VM exits. Table 24-11 lists the controls supported. See Chapter 27 for complete details of how these controls affect VM exits.

Table 24-11. Definitions of VM-Exit Controls

Bit Position(s)	Name	Description
2	Save debug controls	This control determines whether DR7 and the IA32_DEBUGCTL MSR are saved on VM exit. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
9	Host address-space size	On processors that support Intel 64 architecture, this control determines whether a logical processor is in 64-bit mode after the next VM exit. Its value is loaded into CS.L, IA32_EFER.LME, and IA32_EFER.LMA on every VM exit. ¹ This control must be 0 on processors that do not support Intel 64 architecture.
12	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM exit.
15	Acknowledge interrupt on exit	This control affects VM exits due to external interrupts: <ul style="list-style-type: none"> ▪ If such a VM exit occurs and this control is 1, the logical processor acknowledges the interrupt controller, acquiring the interrupt's vector. The vector is stored in the VM-exit interruption-information field, which is marked valid. ▪ If such a VM exit occurs and this control is 0, the interrupt is not acknowledged and the VM-exit interruption-information field is marked invalid.
18	Save IA32_PAT	This control determines whether the IA32_PAT MSR is saved on VM exit.
19	Load IA32_PAT	This control determines whether the IA32_PAT MSR is loaded on VM exit.
20	Save IA32_EFER	This control determines whether the IA32_EFER MSR is saved on VM exit.
21	Load IA32_EFER	This control determines whether the IA32_EFER MSR is loaded on VM exit.
22	Save VMX-preemption timer value	This control determines whether the value of the VMX-preemption timer is saved on VM exit.
23	Clear IA32_BNDCFGS	This control determines whether the IA32_BNDCFGS MSR is cleared on VM exit.
24	Conceal VMX from PT	If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM exit or a VMCS packet on an SMM VM exit (see Chapter 35).
25	Clear IA32_RTIT_CTL	This control determines whether the IA32_RTIT_CTL MSR is cleared on VM exit.
28	Load CET state	This control determines whether CET-related MSRs and SPP are loaded on VM exit.
29	Load PKRS	This control determines whether the IA32_PKRS MSR is loaded on VM exit.

NOTES:

1. Since the Intel 64 architecture specifies that IA32_EFER.LMA is always set to the logical-AND of CRO.PG and IA32_EFER.LME, and since CRO.PG is always 1 in VMX root operation, IA32_EFER.LMA is always identical to IA32_EFER.LME in VMX root operation.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_EXIT_CTL5 and IA32_VMX_TRUE_EXIT_CTL5 (see Appendix A.4) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 26.2.1.2).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8, 10, 11, 13, 14, 16, and 17. The VMX capability MSR IA32_VMX_EXIT_CTL5 always reports that these bits must be 1.

Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_EXIT_CTL5 MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

24.7.2 VM-Exit Controls for MSRs

A VMM may specify lists of MSRs to be stored and loaded on VM exits. The following VM-exit control fields determine how MSRs are stored on VM exits:

- **VM-exit MSR-store count** (32 bits). This field specifies the number of MSRs to be stored on VM exit. It is recommended that this count not exceed 512.¹ Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.
- **VM-exit MSR-store address** (64 bits). This field contains the physical address of the VM-exit MSR-store area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-store count. The format of each entry is given in Table 24-12. If the VM-exit MSR-store count is not zero, the address must be 16-byte aligned.

Table 24-12. Format of an MSR Entry

Bit Position(s)	Contents
31:0	MSR index
63:32	Reserved
127:64	MSR data

See Section 27.4 for how this area is used on VM exits.

The following VM-exit control fields determine how MSRs are loaded on VM exits:

- **VM-exit MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM exit. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.²
- **VM-exit MSR-load address** (64 bits). This field contains the physical address of the VM-exit MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-load count (see Table 24-12). If the VM-exit MSR-load count is not zero, the address must be 16-byte aligned.

See Section 27.6 for how this area is used on VM exits.

24.8 VM-ENTRY CONTROL FIELDS

The VM-entry control fields govern the behavior of VM entries. They are discussed in Sections 24.8.1 through 24.8.3.

24.8.1 VM-Entry Controls

The **VM-entry controls** constitute a 32-bit vector that governs the basic operation of VM entries. Table 24-13 lists the controls supported. See Chapter 24 for how these controls affect VM entries.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_ENTRY_CTL5 and IA32_VMX_TRUE_ENTRY_CTL5 (see Appendix A.5) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 26.2.1.3).

1. Future implementations may allow more MSRs to be stored reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).
2. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).

Table 24-13. Definitions of VM-Entry Controls

Bit Position(s)	Name	Description
2	Load debug controls	This control determines whether DR7 and the IA32_DEBUGCTL MSR are loaded on VM entry. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
9	IA-32e mode guest	On processors that support Intel 64 architecture, this control determines whether the logical processor is in IA-32e mode after VM entry. Its value is loaded into IA32_EFER.LMA as part of VM entry. ¹ This control must be 0 on processors that do not support Intel 64 architecture.
10	Entry to SMM	This control determines whether the logical processor is in system-management mode (SMM) after VM entry. This control must be 0 for any VM entry from outside SMM.
11	Deactivate dual-monitor treatment	If set to 1, the default treatment of SMIs and SMM is in effect after the VM entry (see Section 34.15.7). This control must be 0 for any VM entry from outside SMM.
13	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM entry.
14	Load IA32_PAT	This control determines whether the IA32_PAT MSR is loaded on VM entry.
15	Load IA32_EFER	This control determines whether the IA32_EFER MSR is loaded on VM entry.
16	Load IA32_BNDCFGS	This control determines whether the IA32_BNDCFGS MSR is loaded on VM entry.
17	Conceal VMX from PT	If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM entry or a VMCS packet on a VM entry that returns from SMM (see Chapter 35).
18	Load IA32_RTIT_CTL	This control determines whether the IA32_RTIT_CTL MSR is loaded on VM entry.
20	Load CET state	This control determines whether CET-related MSRs and SPP are loaded on VM entry.
22	Load PKRS	This control determines whether the IA32_PKRS MSR is loaded on VM entry.

NOTES:

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control. If it is read as 1, every VM exit stores the value of IA32_EFER.LMA into the “IA-32e mode guest” VM-entry control (see Section 27.2).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8 and 12. The VMX capability MSR IA32_VMX_ENTRY_CTLS always reports that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_ENTRY_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

24.8.2 VM-Entry Controls for MSRs

A VMM may specify a list of MSRs to be loaded on VM entries. The following VM-entry control fields manage this functionality:

- **VM-entry MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM entry. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM entry.¹
- **VM-entry MSR-load address** (64 bits). This field contains the physical address of the VM-entry MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-entry MSR-load count. The format of entries is described in Table 24-12. If the VM-entry MSR-load count is not zero, the address must be 16-byte aligned.

1. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).

See Section 26.4 for details of how this area is used on VM entries.

24.8.3 VM-Entry Controls for Event Injection

VM entry can be configured to conclude by delivering an event through the IDT (after all guest state and MSRs have been loaded). This process is called **event injection** and is controlled by the following three VM-entry control fields:

- **VM-entry interruption-information field** (32 bits). This field provides details about the event to be injected. Table 24-14 describes the field.

Table 24-14. Format of the VM-Entry Interruption-Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Reserved 2: Non-maskable interrupt (NMI) 3: Hardware exception (e.g., #PF) 4: Software interrupt (INT <i>n</i>) 5: Privileged software exception (INT1) 6: Software exception (INT3 or INTO) 7: Other event
11	Deliver error code (0 = do not deliver; 1 = deliver)
30:12	Reserved
31	Valid

- The **vector** (bits 7:0) determines which entry in the IDT is used or which other event is injected.
- The **interruption type** (bits 10:8) determines details of how the injection is performed. In general, a VMM should use the type hardware exception for all exceptions **other than** the following:
 - breakpoint exceptions (#BP; a VMM should use the type software exception);
 - overflow exceptions (#OF a VMM should use the use type software exception); and
 - those debug exceptions (#DB) that are generated by INT1 (a VMM should use the use type privileged software exception).¹

The type **other event** is used for injection of events that are not delivered through the IDT.²

- For exceptions, the **deliver-error-code bit** (bit 11) determines whether delivery pushes an error code on the guest stack.
- VM entry injects an event if and only if the **valid bit** (bit 31) is 1. The valid bit in this field is cleared on every VM exit (see Section 27.2).
- **VM-entry exception error code** (32 bits). This field is used if and only if the valid bit (bit 31) and the deliver-error-code bit (bit 11) are both set in the VM-entry interruption-information field.
- **VM-entry instruction length** (32 bits). For injection of events whose type is software interrupt, software exception, or privileged software exception, this field is used to determine the value of RIP that is pushed on the stack.

See Section 26.6 for details regarding the mechanics of event injection, including the use of the interruption type and the VM-entry instruction length.

VM exits clear the valid bit (bit 31) in the VM-entry interruption-information field.

1. The type hardware exception should be used for all other debug exceptions.
 2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with values 1 or 3 for *n*.

24.9 VM-EXIT INFORMATION FIELDS

The VMCS contains a section of fields that contain information about the most recent VM exit.

On some processors, attempts to write to these fields with VMWRITE fail (see “VMWRITE—Write Field to Virtual-Machine Control Structure” in Chapter 30).¹

24.9.1 Basic VM-Exit Information

The following VM-exit information fields provide basic information about a VM exit:

- **Exit reason** (32 bits). This field encodes the reason for the VM exit and has the structure given in Table 24-15.

Table 24-15. Format of Exit Reason

Bit Position(s)	Contents
15:0	Basic exit reason
16	Always cleared to 0
26:17	Not currently defined
27	A VM exit saves this bit as 1 to indicate that the VM exit was incident to enclave mode.
28	Pending MTF VM exit
29	VM exit from VMX root operation
30	Not currently defined
31	VM-entry failure (0 = true VM exit; 1 = VM-entry failure)

- Bits 15:0 provide basic information about the cause of the VM exit (if bit 31 is clear) or of the VM-entry failure (if bit 31 is set). Appendix C enumerates the basic exit reasons.
- Bit 16 is always cleared to 0.
- Bit 27 is set to 1 if the VM exit occurred while the logical processor was in enclave mode.
 - A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1). See Section 27.2.1 for details.
- Bit 28 is set only by an SMM VM exit (see Section 34.15.2) that took priority over an MTF VM exit (see Section 25.5.2) that would have occurred had the SMM VM exit not occurred. See Section 34.15.2.3.
- Bit 29 is set if and only if the processor was in VMX root operation at the time the VM exit occurred. This can happen only for SMM VM exits. See Section 34.15.2.
- Because some VM-entry failures load processor state from the host-state area (see Section 26.8), software must be able to distinguish such cases from true VM exits. Bit 31 is used for that purpose.
- **Exit qualification** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field contains additional information about the cause of VM exits due to the following: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); task switches; INVEPT; INVLPG; INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; and MWAIT. The format of the field depends on the cause of the VM exit. See Section 27.2.1 for details.
- **Guest-linear address** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is used in the following cases:
 - VM exits due to attempts to execute LMSW with a memory operand.

1. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

- VM exits due to attempts to execute INS or OUTS.
- VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions.
- Certain VM exits due to EPT violations

See Section 27.2.1 and Section 34.15.2.3 for details of when and how this field is used.

- **Guest-physical address** (64 bits). This field is used VM exits due to EPT violations and EPT misconfigurations. See Section 27.2.1 for details of when and how this field is used.

24.9.2 Information for VM Exits Due to Vectored Events

Event-specific information is provided for VM exits due to the following vectored events: exceptions (including those generated by the instructions INT3, INTO, INT1, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). This information is provided in the following fields:

- **VM-exit interruption information** (32 bits). This field receives basic information associated with the event causing the VM exit. Table 24-16 describes this field.

Table 24-16. Format of the VM-Exit Interruption-Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Not used 2: Non-maskable interrupt (NMI) 3: Hardware exception 4: Not used 5: Privileged software exception 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
12	NMI unblocking due to IRET
30:13	Not currently defined
31	Valid

- **VM-exit interruption error code** (32 bits). For VM exits caused by hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

Section 27.2.2 provides details of how these fields are saved on VM exits.

24.9.3 Information for VM Exits That Occur During Event Delivery

Additional information is provided for VM exits that occur during event delivery in VMX non-root operation.¹ This information is provided in the following fields:

1. This includes cases in which the event delivery was caused by event injection as part of VM entry; see Section 26.6.1.2.

- **IDT-vectoring information** (32 bits). This field receives basic information associated with the event that was being delivered when the VM exit occurred. Table 24-17 describes this field.

Table 24-17. Format of the IDT-Vectoring Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Not used 2: Non-maskable interrupt (NMI) 3: Hardware exception 4: Software interrupt 5: Privileged software exception 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
30:12	Not currently defined
31	Valid

- **IDT-vectoring error code** (32 bits). For VM exits that occur during delivery of hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

See Section 27.2.4 provides details of how these fields are saved on VM exits.

24.9.4 Information for VM Exits Due to Instruction Execution

The following fields are used for VM exits caused by attempts to execute certain instructions in VMX non-root operation:

- **VM-exit instruction length** (32 bits). For VM exits resulting from instruction execution, this field receives the length in bytes of the instruction whose execution led to the VM exit.¹ See Section 27.2.5 for details of when and how this field is used.
- **VM-exit instruction information** (32 bits). This field is used for VM exits due to attempts to execute `INS`, `INVEPT`, `INVVPID`, `LIDT`, `LGDT`, `LLDT`, `LTR`, `OUTS`, `SIDT`, `SGDT`, `SLDT`, `STR`, `VMCLEAR`, `VMPTRLD`, `VMPTRST`, `VMREAD`, `VMWRITE`, or `VMXON`.² The format of the field depends on the cause of the VM exit. See Section 27.2.5 for details.

The following fields (64 bits each; 32 bits on processors that do not support Intel 64 architecture) are used only for VM exits due to SMIs that arrive immediately after retirement of I/O instructions. They provide information about that I/O instruction:

- **I/O RCX**. The value of RCX before the I/O instruction started.
- **I/O RSI**. The value of RSI before the I/O instruction started.
- **I/O RDI**. The value of RDI before the I/O instruction started.
- **I/O RIP**. The value of RIP before the I/O instruction started (the RIP that addressed the I/O instruction).

1. This field is also used for VM exits that occur during the delivery of a software interrupt or software exception.

2. Whether the processor provides this information on VM exits due to attempts to execute `INS` or `OUTS` can be determined by consulting the VMX capability MSR `IA32_VMX_BASIC` (see Appendix A.1).

24.9.5 VM-Instruction Error Field

The 32-bit **VM-instruction error field** does not provide information about the most recent VM exit. In fact, it is not modified on VM exits. Instead, it provides information about errors encountered by a non-faulting execution of one of the VMX instructions.

24.10 VMCS TYPES: ORDINARY AND SHADOW

Every VMCS is either an **ordinary VMCS** or a **shadow VMCS**. A VMCS's type is determined by the shadow-VMCS indicator in the VMCS region (this is the value of bit 31 of the first 4 bytes of the VMCS region; see Table 24-1): 0 indicates an ordinary VMCS, while 1 indicates a shadow VMCS. Shadow VMCSs are supported only on processors that support the 1-setting of the "VMCS shadowing" VM-execution control (see Section 24.6.2).

A shadow VMCS differs from an ordinary VMCS in two ways:

- An ordinary VMCS can be used for VM entry but a shadow VMCS cannot. Attempts to perform VM entry when the current VMCS is a shadow VMCS fail (see Section 26.1).
- The VMREAD and VMWRITE instructions can be used in VMX non-root operation to access a shadow VMCS but not an ordinary VMCS. This fact results from the following:
 - If the "VMCS shadowing" VM-execution control is 0, execution of the VMREAD and VMWRITE instructions in VMX non-root operation always cause VM exits (see Section 25.1.3).
 - If the "VMCS shadowing" VM-execution control is 1, execution of the VMREAD and VMWRITE instructions in VMX non-root operation can access the VMCS referenced by the VMCS link pointer (see Section 30.3).
 - If the "VMCS shadowing" VM-execution control is 1, VM entry ensures that any VMCS referenced by the VMCS link pointer is a shadow VMCS (see Section 26.3.1.5).

In VMX root operation, both types of VMCSs can be accessed with the VMREAD and VMWRITE instructions.

Software should not modify the shadow-VMCS indicator in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted (see Section 24.11.1). Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

24.11 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES

This section details guidelines that software should observe when using a VMCS and related structures. It also provides descriptions of consequences for failing to follow guidelines.

24.11.1 Software Use of Virtual-Machine Control Structures

To ensure proper processor behavior, software should observe certain guidelines when using an active VMCS.

No VMCS should ever be active on more than one logical processor. If a VMCS is to be "migrated" from one logical processor to another, the first logical processor should execute VMCLEAR for the VMCS (to make it inactive on that logical processor and to ensure that all VMCS data are in memory) before the other logical processor executes VMPTRLD for the VMCS (to make it active on the second logical processor).¹ A VMCS that is made active on more than one logical processor may become **corrupted** (see below).

Software should not modify the shadow-VMCS indicator (see Table 24-1) in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted. Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

Software should use the VMREAD and VMWRITE instructions to access the different fields in the current VMCS (see Section 24.11.2). Software should never access or modify the VMCS data of an active VMCS using ordinary

1. As noted in Section 24.1, execution of the VMPTRLD instruction makes a VMCS is active. In addition, VM entry makes active any shadow VMCS referenced by the VMCS link pointer in the current VMCS. If a shadow VMCS is made active by VM entry, it is necessary to execute VMCLEAR for that VMCS before allowing that VMCS to become active on another logical processor.

memory operations, in part because the format used to store the VMCS data is implementation-specific and not architecturally defined, and also because a logical processor may maintain some VMCS data of an active VMCS on the processor and not in the VMCS region. The following items detail some of the hazards of accessing VMCS data using ordinary memory operations:

- Any data read from a VMCS with an ordinary memory read does not reliably reflect the state of the VMCS. Results may vary from time to time or from logical processor to logical processor.
- Writing to a VMCS with an ordinary memory write is not guaranteed to have a deterministic effect on the VMCS. Doing so may cause the VMCS to become corrupted (see below).

(Software can avoid these hazards by removing any linear-address mappings to a VMCS region before executing a VMPTRLD for that region and by not remapping it until after executing VMCLEAR for that region.)

If a logical processor leaves VMX operation, any VMCSs active on that logical processor may be corrupted (see below). To prevent such corruption of a VMCS that may be used either after a return to VMX operation or on another logical processor, software should execute VMCLEAR for that VMCS before executing the VMXOFF instruction or removing power from the processor (e.g., as part of a transition to the S3 and S4 power states).

This section has identified operations that may cause a VMCS to become corrupted. These operations may cause the VMCS’s data to become undefined. Behavior may be unpredictable if that VMCS used subsequently on any logical processor. The following items detail some hazards of VMCS corruption:

- VM entries may fail for unexplained reasons or may load undesired processor state.
- The processor may not correctly support VMX non-root operation as documented in Chapter 25 and may generate unexpected VM exits.
- VM exits may load undesired processor state, save incorrect state into the VMCS, or cause the logical processor to transition to a shutdown state.

24.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields

Every field of the VMCS is associated with a 32-bit value that is its **encoding**. The encoding is provided in an operand to VMREAD and VMWRITE when software wishes to read or write that field. These instructions fail if given, in 64-bit mode, an operand that sets an encoding bit beyond bit 32. See Chapter 30 for a description of these instructions.

The structure of the 32-bit encodings of the VMCS components is determined principally by the width of the fields and their function in the VMCS. See Table 24-18.

Table 24-18. Structure of VMCS Component Encoding

Bit Position(s)	Contents
0	Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural-width fields
9:1	Index
11:10	Type: 0: control 1: VM-exit information 2: guest state 3: host state
12	Reserved (must be 0)
14:13	Width: 0: 16-bit 1: 64-bit 2: 32-bit 3: natural-width
31:15	Reserved (must be 0)

The following items detail the meaning of the bits in each encoding:

- **Field width.** Bits 14:13 encode the width of the field.
 - A value of 0 indicates a 16-bit field.
 - A value of 1 indicates a 64-bit field.
 - A value of 2 indicates a 32-bit field.
 - A value of 3 indicates a **natural-width** field. Such fields have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

Fields whose encodings use value 1 are specially treated to allow 32-bit software access to all 64 bits of the field. Such access is allowed by defining, for each such field, an encoding that allows direct access to the high 32 bits of the field. See below.

- **Field type.** Bits 11:10 encode the type of VMCS field: control, guest-state, host-state, or VM-exit information. (The last category also includes the VM-instruction error field.)
- **Index.** Bits 9:1 distinguish components with the same field width and type.
- **Access type.** Bit 0 must be 0 for all fields except for 64-bit fields (those with field-width 1; see above). A VMREAD or VMWRITE using an encoding with this bit cleared to 0 accesses the entire field. For a 64-bit field with field-width 1, a VMREAD or VMWRITE using an encoding with this bit set to 1 accesses only the high 32 bits of the field.

Appendix B gives the encodings of all fields in the VMCS.

The following describes the operation of VMREAD and VMWRITE based on processor mode, VMCS-field width, and access type:

- 16-bit fields:
 - A VMREAD returns the value of the field in bits 15:0 of the destination operand; other bits of the destination operand are cleared to 0.
 - A VMWRITE writes the value of bits 15:0 of the source operand into the VMCS field; other bits of the source operand are not used.
- 32-bit fields:
 - A VMREAD returns the value of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
 - A VMWRITE writes the value of bits 31:0 of the source operand into the VMCS field; in 64-bit mode, bits 63:32 of the source operand are not used.
- 64-bit fields and natural-width fields using the full access type outside IA-32e mode.
 - A VMREAD returns the value of bits 31:0 of the field in its destination operand; bits 63:32 of the field are ignored.
 - A VMWRITE writes the value of its source operand to bits 31:0 of the field and clears bits 63:32 of the field.
- 64-bit fields and natural-width fields using the full access type in 64-bit mode (only on processors that support Intel 64 architecture).
 - A VMREAD returns the value of the field in bits 63:0 of the destination operand
 - A VMWRITE writes the value of bits 63:0 of the source operand into the VMCS field.
- 64-bit fields using the high access type.
 - A VMREAD returns the value of bits 63:32 of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
 - A VMWRITE writes the value of bits 31:0 of the source operand to bits 63:32 of the field; in 64-bit mode, bits 63:32 of the source operand are not used.

Software seeking to read a 64-bit field outside IA-32e mode can use VMREAD with the full access type (reading bits 31:0 of the field) and VMREAD with the high access type (reading bits 63:32 of the field); the order of the two VMREAD executions is not important. Software seeking to modify a 64-bit field outside IA-32e mode should first

use VMWRITE with the full access type (establishing bits 31:0 of the field while clearing bits 63:32) and then use VMWRITE with the high access type (establishing bits 63:32 of the field).

24.11.3 Initializing a VMCS

Software should initialize fields in a VMCS (using VMWRITE) before using the VMCS for VM entry. Failure to do so may result in unpredictable behavior; for example, a VM entry may fail for unexplained reasons, or a successful transition (VM entry or VM exit) may load processor state with unexpected values.

It is not necessary to initialize fields that the logical processor will not use. (For example, it is not necessary to initialize the MSR-bitmap address if the “use MSR bitmaps” VM-execution control is 0.)

A processor maintains some VMCS information that cannot be modified with the VMWRITE instruction; this includes a VMCS’s launch state (see Section 24.1). Such information may be stored in the VMCS data portion of a VMCS region. Because the format of this information is implementation-specific, there is no way for software to know, when it first allocates a region of memory for use as a VMCS region, how the processor will determine this information from the contents of the memory region.

In addition to its other functions, the VMCLEAR instruction initializes any implementation-specific information in the VMCS region referenced by its operand. To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD for the first time. (Figure 24-1 illustrates how execution of VMCLEAR puts a VMCS into a well-defined state.)

The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry for the first time.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since “migrating” a VMCS from one logical processor to another requires use of VMCLEAR (see Section 24.11.1), which sets the launch state of the VMCS to “clear”, such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

24.11.4 Software Access to Related Structures

In addition to data in the VMCS region itself, VMX non-root operation can be controlled by data structures that are referenced by pointers in a VMCS (for example, the I/O bitmaps). While the pointers to these data structures are parts of the VMCS, the data structures themselves are not. They are not accessible using VMREAD and VMWRITE but by ordinary memory writes.

Software should ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 24.11.1). **Exceptions are made for the following data structures (subject to detailed discussion in the sections indicated): EPT paging structures and the data structures used to locate SPP vectors (Section 28.3.3); the virtual-APIC page (Section 29.1); the posted interrupt descriptor (Section 29.6); and the virtualization-exception information area (Section 25.5.7.2).**

24.11.5 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region)¹ that the logical processor uses to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer is subject to the limitations that apply to VMCS pointers:

1. The amount of memory required for the VMXON region is the same as that required for a VMCS region. This size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

- The VMXON pointer must be 4-KByte aligned (bits 11:0 must be zero).
- The VMXON pointer must not set any bits beyond the processor's physical-address width.^{1,2}

Before executing VMXON, software should write the VMCS revision identifier (see Section 24.2) to the VMXON region. (Specifically, it should write the 31-bit VMCS revision identifier to bits 30:0 of the first 4 bytes of the VMXON region; bit 31 should be cleared to 0.) It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 24.11.1).

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. If IA32_VMX_BASIC[48] is read as 1, the VMXON pointer must not set any bits in the range 63:32; see Appendix A.1.

27. 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 chapter: Added new Section 25.5.4 "Translation of Guest-Physical Addresses Used by Intel Processor Trace". Typo corrections as needed.

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.
- **ENCLV.** The ENCLV instruction causes a VM exit if the “enable ENCLV 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 ENCLV-exiting bitmap is 1 (see Section 24.6.17).
 - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLV-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

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.

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.
- **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/host 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/host 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
- **TPAUSE.** The TPAUSE instruction causes a VM exit if the “RDTSC exiting” and “enable user wait and pause” VM-execution controls are both 1.
- **UMWAIT.** The UMWAIT instruction causes a VM exit if the “RDTSC exiting” and “enable user wait and pause” VM-execution controls are both 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.

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.

- 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.
 - 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 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 24.6.20).
- **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 24.6.20).

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 INT1, INT3, INTO, BOUND, UD0, UD1, and UD2.¹

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.)

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT n with value 1 or 3 for n .

- **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 26.7.5).

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 and no blocking of events by STI (see Table 24-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.7.6).

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:²

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.

- **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.
 - 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.

2. 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.

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:

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 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:
 - 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, SMSW reads normally from CR0; if every bit is set in the CR0 guest/host mask, SMSW 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.

- **TPAUSE.** Behavior of the TPAUSE instruction is determined first by the setting of the “enable user wait and pause” VM-execution control:
 - If the “enable user wait and pause” VM-execution control is 0, TPAUSE causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
 - If the “enable user wait and pause” VM-execution control is 1, treatment is based on the setting of the “RDTSC exiting” VM-execution control:
 - If the “RDTSC exiting” VM-execution control is 0, the instruction delays for an amount of time called here the **physical delay**. The physical delay is first computed by determining the **virtual delay** (the time to delay relative to the guest’s timestamp counter).
 If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFFCH).
 The physical delay depends upon the settings of the “use TSC offsetting” and “use TSC scaling” VM-execution controls:
 - If either control is 0, the physical delay is the virtual delay.
 - If both controls are 1, the virtual delay is multiplied by 2^{48} (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.
 - If the “RDTSC exiting” VM-execution control is 1, TPAUSE causes a VM exit.
- **UMONITOR.** Behavior of the UMONITOR instruction is determined by the setting of the “enable user wait and pause” VM-execution control:
 - If the “enable user wait and pause” VM-execution control is 0, UMONITOR causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
 - If the “enable user wait and pause” VM-execution control is 1, UMONITOR operates normally.
- **UMWAIT.** Behavior of the UMWAIT instruction is determined first by the setting of the “enable user wait and pause” VM-execution control:
 - If the “enable user wait and pause” VM-execution control is 0, UMWAIT causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
 - If the “enable user wait and pause” VM-execution control is 1, treatment is based on the setting of the “RDTSC exiting” VM-execution control:
 - If the “RDTSC exiting” VM-execution control is 0, and if the instruction causes a delay, the amount of time delayed is called here the **physical delay**. The physical delay is first computed by determining the **virtual delay** (the time to delay relative to the guest’s timestamp counter).
 If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFFCH).
 The physical delay depends upon the settings of the “use TSC offsetting” and “use TSC scaling” VM-execution controls:
 - If either control is 0, the physical delay is the virtual delay.
 - If both controls are 1, the virtual delay is multiplied by 2^{48} (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.
 - If the “RDTSC exiting” VM-execution control is 1, UMWAIT causes a VM exit.

- **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.¹
 - If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), or 83FH (self-IPI MSR), instruction behavior may be 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 24.6.20):
 - 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 24.6.20):
 - 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).

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).

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 , INT1, INT3, INTO, 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 and Section 25.5.4), APIC virtualization (Section 25.5.5), VM functions (Section 25.5.6), and virtualization exceptions (Section 25.5.7).

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 26.7.4) 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 26.6.1), 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 26.6.2), 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 INT1, 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.
- 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 42.2 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 Translation of Guest-Physical Addresses Used by Intel Processor Trace

As described in Chapter 35, Intel® Processor Trace (Intel PT) captures information about software execution using dedicated hardware facilities.

Intel PT can be configured so that the trace output is written to memory using physical addresses. For example, when the ToPA (table of physical addresses) output mechanism is used, the IA32_RTIT_OUTPUT_BASE MSR contains the physical address of the base of the current ToPA. Each entry in that table contains the physical address of an output region in memory. When an output region becomes full, the ToPA output mechanism directs subsequent trace output to the next output region as indicated in the ToPA.

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the logical processor treats the addresses used by Intel PT (the output addresses as well as those used to discover the output addresses) as guest-physical addresses, translating to physical addresses using EPT before trace output is written to memory.

Translating these addresses through EPT implies that the trace-output mechanism may cause EPT violations and VM exits; details are provided in Section 25.5.3.1. Section 25.5.3.2 describes a mechanism that ensures that these VM exits do not cause loss of trace data.

25.5.4.1 Guest-Physical Address Translation for Intel PT: Details

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the addresses used by Intel PT are treated as guest-physical addresses and translated using EPT. These addresses include the addresses of the output regions as well as the addresses of the ToPA entries that contain the output-region addresses.

1. This item includes the cases of an invalid opcode exception—#UD—generated by the UDO, UD1, and UD2 instructions and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.

Translation of accesses by the trace-output process may result in EPT violations or EPT misconfigurations (Section 28.2.3), resulting in VM exits. EPT violations resulting for the trace-output process always cause VM exits and are never converted to virtualization exceptions (Section 25.5.7.1).

If no EPT violation or EPT misconfiguration occurs and if page-modification logging (Section 28.2.6) is enabled, the address of an output region may be added to the page-modification log. If the log is full, a page-modification log-full event occurs, resulting in a VM exit.

If the “virtualize APIC accesses” VM-execution control is 1, a guest-physical address used by the trace-output process may be translated to an address on the APIC-access page. In this case, the access by the trace-output process causes an APIC-access VM exit as discussed in Section 29.4.6.1.

25.5.4.2 Trace-Address Pre-Translation (TAPT)

Because it buffers trace data produced by Intel PT before it is written to memory, the processor ensures that buffered data is not lost when a VM exit disables Intel PT. Specifically, the processor ensures that there is sufficient space left in the current output page for the buffered data. If this were not done, buffered trace data could be lost and the resulting trace corrupted.

To prevent the loss of buffered trace data, the processor uses a mechanism called **trace-address pre-translation (TAPT)**. With TAPT, the processor translates using EPT the guest-physical address of the current output region before that address would be used to write buffered trace data to memory.

Because of TAPT, no translation (and thus no EPT violation) occurs at the time output is written to memory; the writes to memory use translations that were cached as part of TAPT. (The details given in Section 25.5.3.1 apply to TAPT.) TAPT ensures that, if a write to the output region would cause an EPT violation, the resulting VM exit is delivered at the time of TAPT, before the region would be used. This allows software to resolve the EPT violation at that time and ensures that, when it is necessary to write buffered trace data to memory, that data will not be lost due to an EPT violation.

TAPT (and resulting VM exits) may occur at any of the following times:

- When software in VMX non-root operation enables tracing by loading the IA32_RTIT_CTL MSR to set the TraceEn bit, using the WRMSR instruction or the XRSTORS instruction.
Any VM exit resulting from TAPT in this case is trap-like: the WRMSR or XRSTORS completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).
- At an instruction boundary when one output region becomes full and Intel PT transitions to the next output region.
VM exits resulting from TAPT in this case take priority over any pending debug exceptions. Such a VM exit will save information about such exceptions in the guest-state area of the VMCS.
- As part of a VM entry that enables Intel PT. See Section 26.5 for details.

TAPT may translate not only the guest-physical address of the current output region but those of subsequent output regions as well. (Doing so may provide better protection of trace data.) This implies that any VM exits resulting from TAPT may result from the translation of output-region addresses other than that of the current output region.

25.5.5 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.

25.5.6 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 25.5.6.1 explains how VM functions are enabled. Section 25.5.6.2 specifies the behavior of the VMFUNC instruction. Section 25.5.6.3 describes a specific VM function called **EPTP switching**.

25.5.6.1 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).

25.5.6.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (#UD) if the “enable VM functions” VM-execution controls is 0¹ 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.

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 25.5.6.3.

25.5.6.3 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 VM exit;
```

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.


```

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 25.5.7.2);
        FI;
    FI;
FI;

```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

If the "Intel PT uses guest physical addresses" VM-execution control is 1 and IA32_RTIT_CTL.TraceEn = 1, any execution of the EPTP-switching VM function causes a VM exit.¹

As noted in Section 25.5.6.2, 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).² 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:

- If 32-bit paging or 4-level paging³ 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 25.5.7.2);

1. Such a VM exit ensures the proper recording of trace data that might otherwise be lost during the change of EPT paging-structure hierarchy. Software handling the VM exit can change emulate the VM function and then resume the guest.
2. If the "enable VPID" VM-execution control is 0, the current VPID is 0000H; if CR4.PCIDE = 0, the current PCID is 000H.
3. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

25.5.7 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 25.5.7.1 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 25.5.7.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 25.5.7.3 for details.

25.5.7.1 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**.¹ 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:
 - 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;
- the EPT violation **does not result** from the output process of Intel Processor Trace (Section 25.5.4); 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 25.5.7.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

1. 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.

25.5.7.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 24.6.19). 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 24.6.19 and Section 25.5.6.3)

A VMM may allow guest software to access the virtualization-exception information area. If it does, the guest software may modify that memory (e.g., to clear the 32-bit value at offset 4; see Section 25.5.7.1). (This is an exception to the general requirement given in Section 24.11.4.)

25.5.7.3 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 25.5.7.1).

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 27.2.4). 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 unpagged 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 unpagged 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.

28. Updates to Chapter 26, Volume 3C

Change bars show changes to Chapter 26 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

Changes to this chapter: Added details on protection keys for supervisor-mode pages, and Control-flow Enforcement Technology (CET) details where applicable. Typo corrections as needed.

Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. VMLAUNCH can be used only with a VMCS whose launch state is clear and VMRESUME can be used only with a VMCS whose the launch state is launched. VMLAUNCH should be used for the first VM entry after VMCLEAR; VMRESUME should be used for subsequent VM entries with the same VMCS.

Each VM entry performs the following steps in the order indicated:

1. Basic checks are performed to ensure that VM entry can commence (Section 26.1).
2. The control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation and that the VMCS is correctly configured to support the next VM exit (Section 26.2).
3. The following may be performed in parallel or in any order (Section 26.3):
 - The guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures.
 - Processor state is loaded from the guest-state area and based on controls in the VMCS.
 - Address-range monitoring is cleared.
4. MSRs are loaded from the VM-entry MSR-load area (Section 26.4).
5. If VMLAUNCH is being executed, the launch state of the VMCS is set to “launched.”
6. If the “Intel PT uses guest physical addresses” VM-execution control is 1, trace-address pre-translation (TAPT) may occur (see Section 25.5.4 and Section 26.5).
7. An event may be injected in the guest context (Section 26.6).

Steps 1–4 above perform checks that may cause VM entry to fail. Such failures occur in one of the following three ways:

- Some of the checks in Section 26.1 may generate ordinary faults (for example, an invalid-opcode exception). Such faults are delivered normally.
- Some of the checks in Section 26.1 and all the checks in Section 26.2 cause control to pass to the instruction following the VM-entry instruction. The failure is indicated by setting RFLAGS.ZF¹ (if there is a current VMCS) or RFLAGS.CF (if there is no current VMCS). If there is a current VMCS, an error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 30 for the error numbers.
- The checks in Section 26.3 and Section 26.4 cause processor state to be loaded from the host-state area of the VMCS (as would be done on a VM exit). Information about the failure is stored in the VM-exit information fields. See Section 26.8 for details.

EFLAGS.TF = 1 causes a VM-entry instruction to generate a single-step debug exception only if failure of one of the checks in Section 26.1 and Section 26.2 causes control to pass to the following instruction. A VM-entry does not generate a single-step debug exception in any of the following cases: (1) the instruction generates a fault; (2) failure of one of the checks in Section 26.3 or in loading MSRs causes processor state to be loaded from the host-state area of the VMCS; or (3) the instruction passes all checks in Section 26.1, Section 26.2, and Section 26.3 and there is no failure in loading MSRs.

Section 34.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, code running in SMM returns using VM entries instead of the RSM instruction. A VM entry **returns from SMM** if it is executed in SMM and the “entry to SMM” VM-entry control is 0. VM entries that return from SMM differ from ordinary VM entries in ways that are detailed in Section 34.15.4.

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

26.1 BASIC VM-ENTRY CHECKS

Before a VM entry commences, the current state of the logical processor is checked in the following order:

1. If the logical processor is in virtual-8086 mode or compatibility mode, an invalid-opcode exception is generated.
2. If the current privilege level (CPL) is not zero, a general-protection exception is generated.
3. If there is no current VMCS, RFLAGS.CF is set to 1 and control passes to the next instruction.
4. If there is a current VMCS but the current VMCS is a shadow VMCS (see Section 24.10), RFLAGS.CF is set to 1 and control passes to the next instruction.
5. If there is a current VMCS that is not a shadow VMCS, the following conditions are evaluated in order; any of these cause VM entry to fail:
 - a. if there is MOV-SS blocking (see Table 24-3)
 - b. if the VM entry is invoked by VMLAUNCH and the VMCS launch state is not clear
 - c. if the VM entry is invoked by VMRESUME and the VMCS launch state is not launched

If any of these checks fail, RFLAGS.ZF is set to 1 and control passes to the next instruction. An error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 30 for the error numbers.

26.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA

If the checks in Section 26.1 do not cause VM entry to fail, the control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation, that the VMCS is correctly configured to support the next VM exit, and that, after the next VM exit, the processor's state is consistent with the Intel 64 and IA-32 architectures.

VM entry fails if any of these checks fail. When such failures occur, control is passed to the next instruction, RFLAGS.ZF is set to 1 to indicate the failure, and the VM-instruction error field is loaded with an error number that indicates whether the failure was due to the controls or the host-state area (see Chapter 30).

These checks may be performed in any order. Thus, an indication by error number of one cause (for example, host state) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same VMCS. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The checks on the controls and the host-state area are presented in Section 26.2.1 through Section 26.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

26.2.1 Checks on VMX Controls

This section identifies VM-entry checks on the VMX control fields.

26.2.1.1 VM-Execution Control Fields

VM entries perform the following checks on the VM-execution control fields:¹

- Reserved bits in the pin-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.3.1).

1. If the "activate secondary controls" primary processor-based VM-execution control is 0, VM entry operates as if each secondary processor-based VM-execution control were 0.

- Reserved bits in the primary processor-based VM-execution controls must be set properly. Software may consult the VMX capability MSR to determine the proper settings (see Appendix A.3.2).
- If the “activate secondary controls” primary processor-based VM-execution control is 1, reserved bits in the secondary processor-based VM-execution controls must be cleared. Software may consult the VMX capability MSR to determine which bits are reserved (see Appendix A.3.3).
If the “activate secondary controls” primary processor-based VM-execution control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the secondary processor-based VM-execution controls. The logical processor operates as if all the secondary processor-based VM-execution controls were 0.
- The CR3-target count must not be greater than 4. Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC to determine the number of values supported (see Appendix A.6).
- If the “use I/O bitmaps” VM-execution control is 1, bits 11:0 of each I/O-bitmap address must be 0. Neither address should set any bits beyond the processor’s physical-address width.^{1,2}
- If the “use MSR bitmaps” VM-execution control is 1, bits 11:0 of the MSR-bitmap address must be 0. The address should not set any bits beyond the processor’s physical-address width.³
- If the “use TPR shadow” VM-execution control is 1, the virtual-APIC address must satisfy the following checks:
 - Bits 11:0 of the address must be 0.
 - The address should not set any bits beyond the processor’s physical-address width.⁴
 If all of the above checks are satisfied and the “use TPR shadow” VM-execution control is 1, bytes 3:1 of VTPR (see Section 29.1.1) may be cleared (behavior may be implementation-specific).
The clearing of these bytes may occur even if the VM entry fails. This is true either if the failure causes control to pass to the instruction following the VM-entry instruction or if it causes processor state to be loaded from the host-state area of the VMCS.
- If the “use TPR shadow” VM-execution control is 1 and the “virtual-interrupt delivery” VM-execution control is 0, bits 31:4 of the TPR threshold VM-execution control field must be 0.⁵
- The following check is performed if the “use TPR shadow” VM-execution control is 1 and the “virtualize APIC accesses” and “virtual-interrupt delivery” VM-execution controls are both 0: the value of bits 3:0 of the TPR threshold VM-execution control field should not be greater than the value of bits 7:4 of VTPR (see Section 29.1.1).
- If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” VM-execution control must be 0.
- If the “virtual NMIs” VM-execution control is 0, the “NMI-window exiting” VM-execution control must be 0.
- If the “virtualize APIC-accesses” VM-execution control is 1, the APIC-access address must satisfy the following checks:
 - Bits 11:0 of the address must be 0.
 - The address should not set any bits beyond the processor’s physical-address width.⁶
- If the “use TPR shadow” VM-execution control is 0, the following VM-execution controls must also be 0: “virtualize x2APIC mode”, “APIC-register virtualization”, and “virtual-interrupt delivery”.⁷

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. If IA32_VMX_BASIC[48] is read as 1, these addresses must not set any bits in the range 63:32; see Appendix A.1.

3. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

4. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

5. “Virtual-interrupt delivery” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “virtual-interrupt delivery” VM-execution control were 0. See Section 24.6.2.

6. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

7. “Virtualize x2APIC mode” and “APIC-register virtualization” are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if these controls were 0. See Section 24.6.2.

- If the “virtualize x2APIC mode” VM-execution control is 1, the “virtualize APIC accesses” VM-execution control must be 0.
- If the “virtual-interrupt delivery” VM-execution control is 1, the “external-interrupt exiting” VM-execution control must be 1.
- If the “process posted interrupts” VM-execution control is 1, the following must be true:¹
 - The “virtual-interrupt delivery” VM-execution control is 1.
 - The “acknowledge interrupt on exit” VM-exit control is 1.
 - The posted-interrupt notification vector has a value in the range 0–255 (bits 15:8 are all 0).
 - Bits 5:0 of the posted-interrupt descriptor address are all 0.
 - The posted-interrupt descriptor address does not set any bits beyond the processor’s physical-address width.²
- If the “enable VPID” VM-execution control is 1, the value of the VPID VM-execution control field must not be 0000H.³
- If the “enable EPT” VM-execution control is 1, the EPTP VM-execution control field (see Table 24-8 in Section 24.6.11) must satisfy the following checks:⁴
 - The EPT memory type (bits 2:0) must be a value supported by the processor as indicated in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A.10).
 - Bits 5:3 (1 less than the EPT page-walk length) must be 3, indicating an EPT page-walk length of 4; see Section 28.2.2.
 - Bit 6 (enable bit for accessed and dirty flags for EPT) must be 0 if bit 21 of the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A.10) is read as 0, indicating that the processor does not support accessed and dirty flags for EPT.
 - Reserved bits 11:7 and 63:N (where N is the processor’s physical-address width) must all be 0.
- If the “enable PML” VM-execution control is 1, the “enable EPT” VM-execution control must also be 1.⁵ In addition, the PML address must satisfy the following checks:
 - Bits 11:0 of the address must be 0.
 - The address should not set any bits beyond the processor’s physical-address width.
- If either the “unrestricted guest” VM-execution control or the “mode-based execute control for EPT” VM-execution control is 1, the “enable EPT” VM-execution control must also be 1.⁶
- If the “sub-page write permissions for EPT” VM-execution control is 1, the “enable EPT” VM-execution control must also be 1.⁷ In addition, the SPPTP VM-execution control field (see Table 24-10 in Section 24.6.21) must satisfy the following checks:
 - Bits 11:0 of the address must be 0.

1. “Process posted interrupts” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “process posted interrupts” VM-execution control were 0. See Section 24.6.2.

2. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

3. “Enable VPID” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable VPID” VM-execution control were 0. See Section 24.6.2.

4. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.

5. “Enable PML” and “enable EPT” are both secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if both these controls were 0. See Section 24.6.2.

6. All these controls are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if all these controls were 0. See Section 24.6.2.

7. “Sub-page write permissions for EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “sub-page write permissions for EPT” VM-execution control were 0. See Section 24.6.2.

- The address should not set any bits beyond the processor’s physical-address width.
 - If the “enable VM functions” processor-based VM-execution control is 1, reserved bits in the VM-function controls must be clear.¹ Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.11). In addition, the following check is performed based on the setting of bits in the VM-function controls (see Section 24.6.14):
 - If “EPTP switching” VM-function control is 1, the “enable EPT” VM-execution control must also be 1. In addition, the EPTP-list address must satisfy the following checks:
 - Bits 11:0 of the address must be 0.
 - The address must not set any bits beyond the processor’s physical-address width.
- If the “enable VM functions” processor-based VM-execution control is 0, no checks are performed on the VM-function controls.
- If the “VMCS shadowing” VM-execution control is 1, the VMREAD-bitmap and VMWRITE-bitmap addresses must each satisfy the following checks:²
 - Bits 11:0 of the address must be 0.
 - The address must not set any bits beyond the processor’s physical-address width.
 - If the “EPT-violation #VE” VM-execution control is 1, the virtualization-exception information address must satisfy the following checks:³
 - Bits 11:0 of the address must be 0.
 - The address must not set any bits beyond the processor’s physical-address width.
 - If the logical processor is operating with Intel PT enabled (if IA32_RTIT_CTL.TraceEn = 1) at the time of VM entry, the “load IA32_RTIT_CTL” VM-entry control must be 0.
 - If the “Intel PT uses guest physical addresses” VM-execution control is 1, the following controls must also be 1: the “enable EPT” VM-execution control; the “load IA32_RTIT_CTL” VM-entry control; and the “clear IA32_RTIT_CTL” VM-exit control.⁴
 - If the “use TSC scaling” VM-execution control is 1, the TSC-multiplier must not be zero.⁵

26.2.1.2 VM-Exit Control Fields

VM entries perform the following checks on the VM-exit control fields.

- Reserved bits in the VM-exit controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.4).
- If the “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-exit control must also be 0.
- The following checks are performed for the VM-exit MSR-store address if the VM-exit MSR-store count field is non-zero:
 - The lower 4 bits of the VM-exit MSR-store address must be 0. The address should not set any bits beyond the processor’s physical-address width.⁶

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, VM entry functions as if the “enable VM functions” VM-execution control were 0. See Section 24.6.2.

2. “VMCS shadowing” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “VMCS shadowing” VM-execution control were 0. See Section 24.6.2.

3. “EPT-violation #VE” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “EPT-violation #VE” VM-execution control were 0. See Section 24.6.2.

4. “Intel PT uses guest physical addresses” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “Intel PT uses guest physical addresses” VM-execution control were 0. See Section 24.6.2.

5. “Use TSC scaling” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “use TSC scaling” VM-execution control were 0. See Section 24.6.2.

- The address of the last byte in the VM-exit MSR-store area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-exit MSR-store address + (MSR count * 16) - 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

- The following checks are performed for the VM-exit MSR-load address if the VM-exit MSR-load count field is non-zero:
 - The lower 4 bits of the VM-exit MSR-load address must be 0. The address should not set any bits beyond the processor's physical-address width.
 - The address of the last byte in the VM-exit MSR-load area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-exit MSR-load address + (MSR count * 16) - 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)
- If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

26.2.1.3 VM-Entry Control Fields

VM entries perform the following checks on the VM-entry control fields.

- Reserved bits in the VM-entry controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.5).
- Fields relevant to VM-entry event injection must be set properly. These fields are the VM-entry interruption-information field (see Table 24-14 in Section 24.8.3), the VM-entry exception error code, and the VM-entry instruction length. If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the following must hold:
 - The field's interruption type (bits 10:8) is not set to a reserved value. Value 1 is reserved on all logical processors; value 7 (other event) is reserved on logical processors that do not support the 1-setting of the "monitor trap flag" VM-execution control.
 - The field's vector (bits 7:0) is consistent with the interruption type:
 - If the interruption type is non-maskable interrupt (NMI), the vector is 2.
 - If the interruption type is hardware exception, the vector is at most 31.
 - If the interruption type is other event, the vector is 0 (pending MTF VM exit).
 - The field's deliver-error-code bit (bit 11) is 1 if each of the following holds: (1) the interruption type is hardware exception; (2) bit 0 (corresponding to CR0.PE) is set in the CR0 field in the guest-state area; (3) IA32_VMX_BASIC[56] is read as 0 (see Appendix A.1); and (4) the vector indicates one of the following exceptions: #DF (vector 8), #TS (10), #NP (11), #SS (12), #GP (13), #PF (14), or #AC (17).
 - The field's deliver-error-code bit is 0 if any of the following holds: (1) the interruption type is not hardware exception; (2) bit 0 is clear in the CR0 field in the guest-state area; or (3) IA32_VMX_BASIC[56] is read as 0 and the vector is in one of the following ranges: 0-7, 9, 15, 16, or 18-31.
 - Reserved bits in the field (30:12) are 0.
 - If the deliver-error-code bit (bit 11) is 1, bits 31:16 of the VM-entry exception error-code field are 0.
 - If the interruption type is software interrupt, software exception, or privileged software exception, the VM-entry instruction-length field is in the range 0-15. A VM-entry instruction length of 0 is allowed only if IA32_VMX_MISC[30] is read as 1; see Appendix A.6.
- The following checks are performed for the VM-entry MSR-load address if the VM-entry MSR-load count field is non-zero:
 - The lower 4 bits of the VM-entry MSR-load address must be 0. The address should not set any bits beyond the processor's physical-address width.¹

6. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- The address of the last byte in the VM-entry MSR-load area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-entry MSR-load address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

- If the processor is not in SMM, the "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls must be 0.
- The "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls cannot both be 1.

26.2.2 Checks on Host Control Registers, MSRs, and SSP

The following checks are performed on fields in the host-state area that correspond to control registers and MSRs:

- The CR0 field must not set any bit to a value not supported in VMX operation (see Section 23.8).¹
- The CR4 field must not set any bit to a value not supported in VMX operation (see Section 23.8).
- If bit 23 in the CR4 field (corresponding to CET) is 1, bit 16 in the CR0 field (WP) must also be 1.
- On processors that support Intel 64 architecture, the CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width must be 0.^{2,3}
- On processors that support Intel 64 architecture, the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field must each contain a canonical address.
- If the "load IA32_PERF_GLOBAL_CTRL" VM-exit control is 1, bits reserved in the IA32_PERF_GLOBAL_CTRL MSR must be 0 in the field for that register (see Figure 18-3).
- If the "load IA32_PAT" VM-exit control is 1, the value of the field for the IA32_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
- If the "load IA32_EFER" VM-exit control is 1, bits reserved in the IA32_EFER MSR must be 0 in the field for that register. In addition, the values of the LMA and LME bits in the field must each be that of the "host address-space size" VM-exit control.
- If the "load CET state" VM-exit control is 1, the IA32_S_CET field must not set any bits reserved in the IA32_S_CET MSR, and bit 10 (corresponding to SUPPRESS) and bit 11 (TRACKER) in the field cannot both be set.
- If the "load CET state" VM-exit control is 1, bits 1:0 must be 0 in the SSP field.
- If the "load PKRS" VM-exit control is 1, bits 63:32 must be 0 in the IA32_PKRS field.

26.2.3 Checks on Host Segment and Descriptor-Table Registers

The following checks are performed on fields in the host-state area that correspond to segment and descriptor-table registers:

- In the selector field for each of CS, SS, DS, ES, FS, GS and TR, the RPL (bits 1:0) and the TI flag (bit 2) must be 0.
- The selector fields for CS and TR cannot be 0000H.
- The selector field for SS cannot be 0000H if the "host address-space size" VM-exit control is 0.

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

1. The bits corresponding to CR0.NW (bit 29) and CR0.CD (bit 30) are never checked because the values of these bits are not changed by VM exit; see Section 27.5.1.

2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

3. Bit 63 of the CR3 field in the host-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.

- On processors that support Intel 64 architecture, the base-address fields for FS, GS, GDTR, IDTR, and TR must contain canonical addresses.

26.2.4 Checks Related to Address-Space Size

On processors that support Intel 64 architecture, the following checks related to address-space size are performed on VMX controls and fields in the host-state area:

- If the logical processor is outside IA-32e mode (if IA32_EFER.LMA = 0) at the time of VM entry, the following must hold:
 - The “IA-32e mode guest” VM-entry control is 0.
 - The “host address-space size” VM-exit control is 0.
- If the logical processor is in IA-32e mode (if IA32_EFER.LMA = 1) at the time of VM entry, the “host address-space size” VM-exit control must be 1.
- If the “host address-space size” VM-exit control is 0, the following must hold:
 - The “IA-32e mode guest” VM-entry control is 0.
 - Bit 17 of the CR4 field (corresponding to CR4.PCIDE) is 0.
 - Bits 63:32 in the RIP field are 0.
 - If the “load CET state” VM-exit control is 1, bits 63:32 in the IA32_S_CET field and in the SSP field are 0.
- If the “host address-space size” VM-exit control is 1, the following must hold:
 - Bit 5 of the CR4 field (corresponding to CR4.PAE) is 1.
 - The RIP field contains a canonical address.
 - If the “load CET state” VM-exit control is 1, the IA32_S_CET field and the SSP field contain canonical addresses.
- If the “load CET state” VM-exit control is 1, the IA32_INTERRUPT_SSP_TABLE_ADDR field contains a canonical address.

On processors that do not support Intel 64 architecture, checks are performed to ensure that the “IA-32e mode guest” VM-entry control and the “host address-space size” VM-exit control are both 0.

26.3 CHECKING AND LOADING GUEST STATE

If all checks on the VMX controls and the host-state area pass (see Section 26.2), the following operations take place concurrently: (1) the guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures; (2) processor state is loaded from the guest-state area or as specified by the VM-entry control fields; and (3) address-range monitoring is cleared.

Because the checking and the loading occur concurrently, a failure may be discovered only after some state has been loaded. For this reason, the logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 26.8.

26.3.1 Checks on the Guest State Area

This section describes checks performed on fields in the guest-state area. These checks may be performed in any order. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The following subsections reference fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

26.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs

The following checks are performed on fields in the guest-state area corresponding to control registers, debug registers, and MSRs:

- The CR0 field must not set any bit to a value not supported in VMX operation (see Section 23.8). The following are exceptions:
 - Bit 0 (corresponding to CR0.PE) and bit 31 (PG) are not checked if the “unrestricted guest” VM-execution control is 1.¹
 - Bit 29 (corresponding to CR0.NW) and bit 30 (CD) are never checked because the values of these bits are not changed by VM entry; see Section 26.3.2.1.
- If bit 31 in the CR0 field (corresponding to PG) is 1, bit 0 in that field (PE) must also be 1.²
- The CR4 field must not set any bit to a value not supported in VMX operation (see Section 23.8).
- If bit 23 in the CR4 field (corresponding to CET) is 1, bit 16 in the CR0 field (WP) must also be 1.
- If the “load debug controls” VM-entry control is 1, bits reserved in the IA32_DEBUGCTL MSR must be 0 in the field for that register. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally.
- The following checks are performed on processors that support Intel 64 architecture:
 - If the “IA-32e mode guest” VM-entry control is 1, bit 31 in the CR0 field (corresponding to CR0.PG) and bit 5 in the CR4 field (corresponding to CR4.PAE) must each be 1.³
 - If the “IA-32e mode guest” VM-entry control is 0, bit 17 in the CR4 field (corresponding to CR4.PCIDE) must be 0.
 - The CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width are 0.^{4,5}
 - If the “load debug controls” VM-entry control is 1, bits 63:32 in the DR7 field must be 0. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally (if they supported Intel 64 architecture).
 - The IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field must each contain a canonical address.
 - If the “load CET state” VM-entry control is 1, the IA32_S_CET field and the IA32_INTERRUPT_SSP_TABLE_ADDR field must contain canonical addresses.
- If the “load IA32_PERF_GLOBAL_CTRL” VM-entry control is 1, bits reserved in the IA32_PERF_GLOBAL_CTRL MSR must be 0 in the field for that register (see Figure 18-3).
- If the “load IA32_PAT” VM-entry control is 1, the value of the field for the IA32_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
- If the “load IA32_EFER” VM-entry control is 1, the following checks are performed on the field for the IA32_EFER MSR:
 - Bits reserved in the IA32_EFER MSR must be 0.
 - Bit 10 (corresponding to IA32_EFER.LMA) must equal the value of the “IA-32e mode guest” VM-entry control. It must also be identical to bit 8 (LME) if bit 31 in the CR0 field (corresponding to CR0.PG) is 1.⁶

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

3. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

4. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

5. Bit 63 of the CR3 field in the guest-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.

- If the “load IA32_BNDCFGS” VM-entry control is 1, the following checks are performed on the field for the IA32_BNDCFGS MSR:
 - Bits reserved in the IA32_BNDCFGS MSR must be 0.
 - The linear address in bits 63:12 must be canonical.
- If the “load IA32_RTIT_CTL” VM-entry control is 1, bits reserved in the IA32_RTIT_CTL MSR must be 0 in the field for that register (see Table 35-6).
- If the “load CET state” VM-entry control is 1, the IA32_S_CET field must not set any bits reserved in the IA32_S_CET MSR, and bit 10 (corresponding to SUPPRESS) and bit 11 (TRACKER) of the field cannot both be set.
- If the “load PKRS” VM-entry control is 1, bits 63:32 must be 0 in the IA32_PKRS field.

26.3.1.2 Checks on Guest Segment Registers

This section specifies the checks on the fields for CS, SS, DS, ES, FS, GS, TR, and LDTR. The following terms are used in defining these checks:

- The guest will be **virtual-8086** if the VM flag (bit 17) is 1 in the RFLAGS field in the guest-state area.
- The guest will be **IA-32e mode** if the “IA-32e mode guest” VM-entry control is 1. (This is possible only on processors that support Intel 64 architecture.)
- Any one of these registers is said to be **usable** if the unusable bit (bit 16) is 0 in the access-rights field for that register.

The following are the checks on these fields:

- Selector fields.
 - TR. The TI flag (bit 2) must be 0.
 - LDTR. If LDTR is usable, the TI flag (bit 2) must be 0.
 - SS. If the guest will not be virtual-8086 and the “unrestricted guest” VM-execution control is 0, the RPL (bits 1:0) must equal the RPL of the selector field for CS.¹
- Base-address fields.
 - CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the address must be the selector field shifted left 4 bits (multiplied by 16).
 - The following checks are performed on processors that support Intel 64 architecture:
 - TR, FS, GS. The address must be canonical.
 - LDTR. If LDTR is usable, the address must be canonical.
 - CS. Bits 63:32 of the address must be zero.
 - SS, DS, ES. If the register is usable, bits 63:32 of the address must be zero.
- Limit fields for CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the field must be 0000FFFFH.
- Access-rights fields.
 - CS, SS, DS, ES, FS, GS.
 - If the guest will be virtual-8086, the field must be 000000F3H. This implies the following:
 - Bits 3:0 (Type) must be 3, indicating an expand-up read/write accessed data segment.
 - Bit 4 (S) must be 1.
 - Bits 6:5 (DPL) must be 3.

6. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CRO.PG must be 1 in VMX operation, bit 31 in the CRO field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

- Bit 7 (P) must be 1.
- Bits 11:8 (reserved), bit 12 (software available), bit 13 (reserved/L), bit 14 (D/B), bit 15 (G), bit 16 (unusable), and bits 31:17 (reserved) must all be 0.
- If the guest will not be virtual-8086, the different sub-fields are considered separately:
 - Bits 3:0 (Type).
 - CS. The values allowed depend on the setting of the “unrestricted guest” VM-execution control:
 - If the control is 0, the Type must be 9, 11, 13, or 15 (accessed code segment).
 - If the control is 1, the Type must be either 3 (read/write accessed expand-up data segment) or one of 9, 11, 13, and 15 (accessed code segment).
 - SS. If SS is usable, the Type must be 3 or 7 (read/write, accessed data segment).
 - DS, ES, FS, GS. The following checks apply if the register is usable:
 - Bit 0 of the Type must be 1 (accessed).
 - If bit 3 of the Type is 1 (code segment), then bit 1 of the Type must be 1 (readable).
 - Bit 4 (S). If the register is CS or if the register is usable, S must be 1.
 - Bits 6:5 (DPL).
 - CS.
 - If the Type is 3 (read/write accessed expand-up data segment), the DPL must be 0. The Type can be 3 only if the “unrestricted guest” VM-execution control is 1.
 - If the Type is 9 or 11 (non-conforming code segment), the DPL must equal the DPL in the access-rights field for SS.
 - If the Type is 13 or 15 (conforming code segment), the DPL cannot be greater than the DPL in the access-rights field for SS.
 - SS.
 - If the “unrestricted guest” VM-execution control is 0, the DPL must equal the RPL from the selector field.
 - The DPL must be 0 either if the Type in the access-rights field for CS is 3 (read/write accessed expand-up data segment) or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.¹
 - DS, ES, FS, GS. The DPL cannot be less than the RPL in the selector field if (1) the “unrestricted guest” VM-execution control is 0; (2) the register is usable; and (3) the Type in the access-rights field is in the range 0 – 11 (data segment or non-conforming code segment).
 - Bit 7 (P). If the register is CS or if the register is usable, P must be 1.
 - Bits 11:8 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
 - Bit 14 (D/B). For CS, D/B must be 0 if the guest will be IA-32e mode and the L bit (bit 13) in the access-rights field is 1.
 - Bit 15 (G). The following checks apply if the register is CS or if the register is usable:
 - If any bit in the limit field in the range 11:0 is 0, G must be 0.
 - If any bit in the limit field in the range 31:20 is 1, G must be 1.
 - Bits 31:17 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
- TR. The different sub-fields are considered separately:

1. The following apply if either the “unrestricted guest” VM-execution control or bit 31 of the primary processor-based VM-execution controls is 0: (1) bit 0 in the CR0 field must be 1 if the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation; and (2) the Type in the access-rights field for CS cannot be 3.

- Bits 3:0 (Type).
 - If the guest will not be IA-32e mode, the Type must be 3 (16-bit busy TSS) or 11 (32-bit busy TSS).
 - If the guest will be IA-32e mode, the Type must be 11 (64-bit busy TSS).
- Bit 4 (S). S must be 0.
- Bit 7 (P). P must be 1.
- Bits 11:8 (reserved). These bits must all be 0.
- Bit 15 (G).
 - If any bit in the limit field in the range 11:0 is 0, G must be 0.
 - If any bit in the limit field in the range 31:20 is 1, G must be 1.
- Bit 16 (Unusable). The unusable bit must be 0.
- Bits 31:17 (reserved). These bits must all be 0.
- LDTR. The following checks on the different sub-fields apply only if LDTR is usable:
 - Bits 3:0 (Type). The Type must be 2 (LDT).
 - Bit 4 (S). S must be 0.
 - Bit 7 (P). P must be 1.
 - Bits 11:8 (reserved). These bits must all be 0.
 - Bit 15 (G).
 - If any bit in the limit field in the range 11:0 is 0, G must be 0.
 - If any bit in the limit field in the range 31:20 is 1, G must be 1.
 - Bits 31:17 (reserved). These bits must all be 0.

26.3.1.3 Checks on Guest Descriptor-Table Registers

The following checks are performed on the fields for GDTR and IDTR:

- On processors that support Intel 64 architecture, the base-address fields must contain canonical addresses.
- Bits 31:16 of each limit field must be 0.

26.3.1.4 Checks on Guest RIP, RFLAGS, and SSP

The following checks are performed on fields in the guest-state area corresponding to RIP, RFLAGS, and SSP (shadow-stack pointer):

- RIP. The following checks are performed on processors that support Intel 64 architecture:
 - Bits 63:32 must be 0 if the “IA-32e mode guest” VM-entry control is 0 or if the L bit (bit 13) in the access-rights field for CS is 0.
 - If the processor supports $N < 64$ linear-address bits, bits 63:N must be identical if the “IA-32e mode guest” VM-entry control is 1 and the L bit in the access-rights field for CS is 1.¹ (No check applies if the processor supports 64 linear-address bits.)
- RFLAGS.
 - Reserved bits 63:22 (bits 31:22 on processors that do not support Intel 64 architecture), bit 15, bit 5 and bit 3 must be 0 in the field, and reserved bit 1 must be 1.
 - The VM flag (bit 17) must be 0 either if the “IA-32e mode guest” VM-entry control is 1 or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.²

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- The IF flag (RFLAGS[bit 9]) must be 1 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) is external interrupt.
- **SSP.** The following checks are performed if the “load CET state” VM-entry control is 1
 - Bits 1:0 must be 0.
 - If the processor supports the Intel 64 architecture, bits 63:N must be identical, where N is the CPU’s maximum linear-address width. (This check does not apply if the processor supports 64 linear-address bits.)

26.3.1.5 Checks on Guest Non-Register State

The following checks are performed on fields in the guest-state area corresponding to non-register state:

- **Activity state.**
 - The activity-state field must contain a value in the range 0 – 3, indicating an activity state supported by the implementation (see Section 24.4.2). Future processors may include support for other activity states. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine what activity states are supported.
 - The activity-state field must not indicate the HLT state if the DPL (bits 6:5) in the access-rights field for SS is not 0.¹
 - The activity-state field must indicate the active state if the interruptibility-state field indicates blocking by either MOV-SS or by STI (if either bit 0 or bit 1 in that field is 1).
 - If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the interruption to be delivered (as defined by interruption type and vector) must not be one that would normally be blocked while a logical processor is in the activity state corresponding to the contents of the activity-state field. The following items enumerate the interruptions (as specified in the VM-entry interruption-information field) whose injection is allowed for the different activity states:
 - Active. Any interruption is allowed.
 - HLT. The only events allowed are the following:
 - Those with interruption type external interrupt or non-maskable interrupt (NMI).
 - Those with interruption type hardware exception and vector 1 (debug exception) or vector 18 (machine-check exception).
 - Those with interruption type other event and vector 0 (pending MTF VM exit).
- See Table 24-14 in Section 24.8.3 for details regarding the format of the VM-entry interruption-information field.
- Shutdown. Only NMIs and machine-check exceptions are allowed.
 - Wait-for-SIPI. No interruptions are allowed.
 - The activity-state field must not indicate the wait-for-SIPI state if the “entry to SMM” VM-entry control is 1.
 - **Interruptibility state.**
 - The reserved bits (bits 31:5) must be 0.
 - The field cannot indicate blocking by both STI and MOV SS (bits 0 and 1 cannot both be 1).
 - Bit 0 (blocking by STI) must be 0 if the IF flag (bit 9) is 0 in the RFLAGS field.
 - Bit 0 (blocking by STI) and bit 1 (blocking by MOV-SS) must both be 0 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) in that field has value 0, indicating external interrupt, or value 2, indicating non-maskable interrupt (NMI).
 - Bit 2 (blocking by SMI) must be 0 if the processor is not in SMM.

2. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CRO.PE must be 1 in VMX operation, bit 0 in the CRO field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

1. As noted in Section 24.4.1, SS.DPL corresponds to the logical processor’s current privilege level (CPL).

- Bit 2 (blocking by SMI) must be 1 if the “entry to SMM” VM-entry control is 1.
- Bit 3 (blocking by NMI) must be 0 if the “virtual NMIs” VM-execution control is 1, the valid bit (bit 31) in the VM-entry interruption-information field is 1, and the interruption type (bits 10:8) in that field has value 2 (indicating NMI).
- If bit 4 (enclave interruption) is 1, bit 1 (blocking by MOV-SS) must be 0 and the processor must support for SGX by enumerating CPUID.(EAX=07H,ECX=0):EBX.SGX[bit 2] as 1.

NOTE

If the “virtual NMIs” VM-execution control is 0, there is no requirement that bit 3 be 0 if the valid bit in the VM-entry interruption-information field is 1 and the interruption type in that field has value 2.

- Pending debug exceptions.
 - Bits 11:4, bit 13, bit 15, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0.
 - The following checks are performed if any of the following holds: (1) the interruptibility-state field indicates blocking by STI (bit 0 in that field is 1); (2) the interruptibility-state field indicates blocking by MOV SS (bit 1 in that field is 1); or (3) the activity-state field indicates HLT:
 - Bit 14 (BS) must be 1 if the TF flag (bit 8) in the RFLAGS field is 1 and the BTF flag (bit 1) in the IA32_DEBUGCTL field is 0.
 - Bit 14 (BS) must be 0 if the TF flag (bit 8) in the RFLAGS field is 0 or the BTF flag (bit 1) in the IA32_DEBUGCTL field is 1.
 - The following checks are performed if bit 16 (RTM) is 1:
 - Bits 11:0, bits 15:13, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0; bit 12 must be 1.
 - The processor must support for RTM by enumerating CPUID.(EAX=07H,ECX=0):EBX[bit 11] as 1.
 - The interruptibility-state field must not indicate blocking by MOV SS (bit 1 in that field must be 0).
- VMCS link pointer. The following checks apply if the field contains a value other than FFFFFFFF_FFFFFFFFH:
 - Bits 11:0 must be 0.
 - Bits beyond the processor’s physical-address width must be 0.^{1,2}
 - The 4 bytes located in memory referenced by the value of the field (as a physical address) must satisfy the following:
 - Bits 30:0 must contain the processor’s VMCS revision identifier (see Section 24.2).³
 - Bit 31 must contain the setting of the “VMCS shadowing” VM-execution control.⁴ This implies that the referenced VMCS is a shadow VMCS (see Section 24.10) if and only if the “VMCS shadowing” VM-execution control is 1.
 - If the processor is not in SMM or the “entry to SMM” VM-entry control is 1, the field must not contain the current VMCS pointer.
 - If the processor is in SMM and the “entry to SMM” VM-entry control is 0, the field must differ from the executive-VMCS pointer.

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. If IA32_VMX_BASIC[48] is read as 1, this field must not set any bits in the range 63:32; see Appendix A.1.

3. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.

4. “VMCS shadowing” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “VMCS shadowing” VM-execution control were 0. See Section 24.6.2.

26.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries

If $CR0.PG = 1$, $CR4.PAE = 1$, and $IA32_EFER.LME = 0$, the logical processor uses **PAE paging** (see Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).¹ When PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs.

A VM entry is to a guest that uses PAE paging if (1) bit 31 (corresponding to CR0.PG) is set in the CR0 field in the guest-state area; (2) bit 5 (corresponding to CR4.PAE) is set in the CR4 field; and (3) the "IA-32e mode guest" VM-entry control is 0. Such a VM entry checks the validity of the PDPTEs:

- If the "enable EPT" VM-execution control is 0, VM entry checks the validity of the PDPTEs referenced by the CR3 field in the guest-state area if either (1) PAE paging was not in use before the VM entry; or (2) the value of CR3 is changing as a result of the VM entry. VM entry may check their validity even if neither (1) nor (2) hold.²
- If the "enable EPT" VM-execution control is 1, VM entry checks the validity of the PDPTE fields in the guest-state area (see Section 24.4.2).

A VM entry to a guest that does not use PAE paging does not check the validity of any PDPTEs.

A VM entry that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use.³ If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), the VM entry fails.

26.3.2 Loading Guest State

Processor state is updated on VM entries in the following ways:

- Some state is loaded from the guest-state area.
- Some state is determined by VM-entry controls.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order and in parallel with the checking of VMCS contents (see Section 26.3.1).

The loading of guest state is detailed in Section 26.3.2.1 to Section 26.3.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

In addition to the state loading described in this section, VM entries may load MSRs from the VM-entry MSR-load area (see Section 26.4). This loading occurs only after the state loading described in this section and the checking of VMCS contents described in Section 26.3.1.

26.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs

The following items describe how guest control registers, debug registers, and MSRs are loaded on VM entry:

- CR0 is loaded from the CR0 field with the exception of the following bits, which are never modified on VM entry: ET (bit 4); reserved bits 15:6, 17, and 28:19; NW (bit 29) and CD (bit 30).⁴ The values of these bits in the CR0 field are ignored.
- CR3 and CR4 are loaded from the CR3 field and the CR4 field, respectively.

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine the number physical-address bits supported by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "enable EPT" VM-execution control were 0. See Section 24.6.2.

3. This implies that (1) bits 11:9 in each PDPTE are ignored; and (2) if bit 0 (present) is clear in one of the PDPTes, bits 63:1 of that PDPTE are ignored.

4. Bits 15:6, bit 17, and bit 28:19 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. Bits 15:6, bit 17, and bit 28:19 of CR0 are always 0 and CR0.ET is always 1.

- If the “load debug controls” VM-entry control is 1, DR7 is loaded from the DR7 field with the exception that bit 12 and bits 15:14 are always 0 and bit 10 is always 1. The values of these bits in the DR7 field are ignored. The first processors to support the virtual-machine extensions supported only the 1-setting of the “load debug controls” VM-entry control and thus always loaded DR7 from the DR7 field.
 - The following describes how certain MSRs are loaded using fields in the guest-state area:
 - If the “load debug controls” VM-entry control is 1, the IA32_DEBUGCTL MSR is loaded from the IA32_DEBUGCTL field. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always loaded the IA32_DEBUGCTL MSR from the IA32_DEBUGCTL field.
 - The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since this field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
 - The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
 - The following are performed on processors that support Intel 64 architecture:
 - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 26.3.2.2).
 - If the “load IA32_EFER” VM-entry control is 0, bits in the IA32_EFER MSR are modified as follows:
 - IA32_EFER.LMA is loaded with the setting of the “IA-32e mode guest” VM-entry control.
 - If CR0 is being loaded so that CR0.PG = 1, IA32_EFER.LME is also loaded with the setting of the “IA-32e mode guest” VM-entry control.¹ Otherwise, IA32_EFER.LME is unmodified.
 - See below for the case in which the “load IA32_EFER” VM-entry control is 1
 - If the “load IA32_PERF_GLOBAL_CTRL” VM-entry control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field.
 - If the “load IA32_PAT” VM-entry control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field.
 - If the “load IA32_EFER” VM-entry control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field.
 - If the “load IA32_BNDCFGS” VM-entry control is 1, the IA32_BNDCFGS MSR is loaded from the IA32_BNDCFGS field.
 - If the “load IA32_RTIT_CTL” VM-entry control is 1, the IA32_RTIT_CTL MSR is loaded from the IA32_RTIT_CTL field.
 - If the “load CET” VM-entry control is 1, the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are loaded from the IA32_S_CET field and the IA32_INTERRUPT_SSP_TABLE_ADDR field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
 - If the “load PKRS” VM-entry control is 1, the IA32_PKRS MSR is loaded from the IA32_PKRS field.
- With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-entry MSR-load area. See Section 26.4.
- The SMBASE register is unmodified by all VM entries except those that return from SMM.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, VM entry must be loading CR0 so that CR0.PG = 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

26.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers

For each of CS, SS, DS, ES, FS, GS, TR, and LDTR, fields are loaded from the guest-state area as follows:

- The unusable bit is loaded from the access-rights field. This bit can never be set for TR (see Section 26.3.1.2). If it is set for one of the other registers, the following apply:
 - For each of CS, SS, DS, ES, FS, and GS, uses of the segment cause faults (general-protection exception or stack-fault exception) outside 64-bit mode, just as they would had the segment been loaded using a null selector. This bit does not cause accesses to fault in 64-bit mode.
 - If this bit is set for LDTR, uses of LDTR cause general-protection exceptions in all modes, just as they would had LDTR been loaded using a null selector.
- If this bit is clear for any of CS, SS, DS, ES, FS, GS, TR, and LDTR, a null selector value does not cause a fault (general-protection exception or stack-fault exception).
- TR. The selector, base, limit, and access-rights fields are loaded.
- CS.
 - The following fields are always loaded: selector, base address, limit, and (from the access-rights field) the L, D, and G bits.
 - For the other fields, the unusable bit of the access-rights field is consulted:
 - If the unusable bit is 0, all of the access-rights field is loaded.
 - If the unusable bit is 1, the remainder of CS access rights are undefined after VM entry.
- SS, DS, ES, FS, GS, and LDTR.
 - The selector fields are loaded.
 - For the other fields, the unusable bit of the corresponding access-rights field is consulted:
 - If the unusable bit is 0, the base-address, limit, and access-rights fields are loaded.
 - If the unusable bit is 1, the base address, the segment limit, and the remainder of the access rights are undefined after VM entry with the following exceptions:
 - Bits 3:0 of the base address for SS are cleared to 0.
 - SS.DPL is always loaded from the SS access-rights field. This will be the current privilege level (CPL) after the VM entry completes.
 - SS.B is always set to 1.
 - The base addresses for FS and GS are loaded from the corresponding fields in the VMCS. On processors that support Intel 64 architecture, the values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
 - On processors that support Intel 64 architecture, the base address for LDTR is set to an undefined but canonical value.
 - On processors that support Intel 64 architecture, bits 63:32 of the base addresses for SS, DS, and ES are cleared to 0.

GDTR and IDTR are loaded using the base and limit fields.

26.3.2.3 Loading Guest RIP, RSP, RFLAGS, and SSP

RSP, RIP, and RFLAGS are loaded from the RSP field, the RIP field, and the RFLAGS field, respectively.

If the “load CET” VM-entry control is 1, SSP (shadow-stack pointer) is loaded from the SSP field.

The following items regard the upper 32 bits of these fields on VM entries that are not to 64-bit mode:

- Bits 63:32 of RSP are undefined outside 64-bit mode. Thus, a logical processor may ignore the contents of bits 63:32 of the RSP field on VM entries that are not to 64-bit mode.
- As noted in Section 26.3.1.4, bits 63:32 of the RIP and RFLAGS fields must be 0 on VM entries that are not to 64-bit mode. (The same is true for SSP for VM entries that are not to 64-bit mode when the “load CET” VM-entry control is 1.)

26.3.2.4 Loading Page-Directory-Pointer-Table Entries

As noted in Section 26.3.1.6, the logical processor uses PAE paging if $CR0.PG = 1$, $CR4.PAE = 1$, and $IA32_EFER.LME = 0$. A VM entry to a guest that uses PAE paging loads the PDPTEs into internal, non-architectural registers based on the setting of the “enable EPT” VM-execution control:

- If the control is 0, the PDPTEs are loaded from the page-directory-pointer table referenced by the physical address in the value of CR3 being loaded by the VM entry (see Section 26.3.2.1). The values loaded are treated as physical addresses in VMX non-root operation.
- If the control is 1, the PDPTEs are loaded from corresponding fields in the guest-state area (see Section 24.4.2). The values loaded are treated as guest-physical addresses in VMX non-root operation.

26.3.2.5 Updating Non-Register State

Section 28.3 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM entries invalidate cached mappings:

- If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP).
- VM entries are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

If the “virtual-interrupt delivery” VM-execution control is 1, VM entry loads the values of RVI and SVI from the guest interrupt-status field in the VMCS (see Section 24.4.2). After doing so, the logical processor first causes PPR virtualization (Section 29.1.3) and then evaluates pending virtual interrupts (Section 29.2.1).

If a virtual interrupt is recognized, it may be delivered in VMX non-root operation immediately after VM entry (including any specified event injection) completes; see Section 26.7.5. See Section 29.2.2 for details regarding the delivery of virtual interrupts.

26.3.3 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 8.10.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. VM entries clear any address-range monitoring that may be in effect.

26.4 LOADING MSRS

VM entries may load MSRs from the VM-entry MSR-load area (see Section 24.8.2). Specifically each entry in that area (up to the number specified in the VM-entry MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.¹

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101 (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM entry did not commence in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM entries for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

1. Because attempts to modify the value of IA32_EFER.LMA by WRMSR are ignored, attempts to modify it using the VM-entry MSR-load area are also ignored.

- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.¹

The VM entry fails if processing fails for any entry. The logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 26.8.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM entry, the logical processor will not use any translations that were cached before the transition.

26.5 TRACE-ADDRESS PRE-TRANSLATION (TAPT)

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the addresses used by Intel PT are treated as guest-physical addresses, and these are translated to physical addresses using EPT.

VM entry uses **trace-address pre-translation (TAPT)** to prevent buffered trace data from being lost due to an EPT violation; see Section 25.5.3.2. VM entry uses TAPT only if Intel PT will be enabled following VM entry (IA32_RTIT_CTL.TraceEn = 1) and only if the “Intel PT uses guest physical addresses” VM-execution control is 1

As noted in Section 25.5.4, TAPT may cause a VM exit due to an EPT violation, EPT misconfiguration, page-modification log-full event, or APIC access. If such a VM exit occurs as a result of TAPT during VM entry, the VM exit operates as if it had occurred in VMX non-root operation after the VM entry completed (in the guest context).

If TAPT during VM entry causes a VM exit, the VM entry does not perform event injection (Section 26.6), even if the valid bit in the VM-entry interruption-information field is 1. Such VM exits save the contents of VM-entry interruption-information and VM-entry exception error code fields into the IDT-vectoring information and IDT-vectoring error code fields, respectively.

26.6 EVENT INJECTION

If the valid bit in the VM-entry interruption-information field (see Section 24.8.3) is 1, VM entry causes an event to be delivered (or made pending) after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.

- If the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception), the event is delivered as described in Section 26.6.1.
- If the interruption type in the field is 7 (other event) and the vector field is 0, an MTF VM exit is pending after VM entry. See Section 26.6.2.

26.6.1 Vectored-Event Injection

VM entry delivers an injected vectored event within the guest context established by VM entry. This means that delivery occurs after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.² The event is delivered using the vector in that field to select a descriptor in the IDT. Since event injection occurs after loading IDTR from the guest-state area, this is the guest IDT.

Section 26.6.1.1 provides details of vectored-event injection. In general, the event is delivered exactly as if it had been generated normally.

1. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. If VM entry has established CR0.PG = 1, the IA32_EFER MSR should not be included in the VM-entry MSR-load area for the purpose of modifying the LME bit.

2. This does not imply that injection of an exception or interrupt will cause a VM exit due to the settings of VM-execution control fields (such as the exception bitmap) that would cause a VM exit if the event had occurred in VMX non-root operation. In contrast, a nested exception encountered during event delivery may cause a VM exit; see Section 26.6.1.1.

If event delivery encounters a nested exception (for example, a general-protection exception because the vector indicates a descriptor beyond the IDT limit), the exception bitmap is consulted using the vector of that exception:

- If the bit for the nested exception is 0, the nested exception is delivered normally. If the nested exception is benign, it is delivered through the IDT. If it is contributory or a page fault, a double fault may be generated, depending on the nature of the event whose delivery encountered the nested exception. See Chapter 6, “Interrupt 8—Double Fault Exception (#DF)” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.¹
- If the bit for the nested exception is 1, a VM exit occurs. Section 26.6.1.2 details cases in which event injection causes a VM exit.

26.6.1.1 Details of Vectored-Event Injection

The event-injection process is controlled by the contents of the VM-entry interruption information field (format given in Table 24-14), the VM-entry exception error-code field, and the VM-entry instruction-length field. The following items provide details of the process:

- The value pushed on the stack for RFLAGS is generally that which was loaded from the guest-state area. The value pushed for the RF flag is not modified based on the type of event being delivered. However, the pushed value of RFLAGS may be modified if a software interrupt is being injected into a guest that will be in virtual-8086 mode (see below). After RFLAGS is pushed on the stack, the value in the RFLAGS register is modified as is done normally when delivering an event through the IDT.
- The instruction pointer that is pushed on the stack depends on the type of event and whether nested exceptions occur during its delivery. The term **current guest RIP** refers to the value to be loaded from the guest-state area. The value pushed is determined as follows:²
 - If VM entry successfully injects (with no nested exception) an event with interruption type external interrupt, NMI, or hardware exception, the current guest RIP is pushed on the stack.
 - If VM entry successfully injects (with no nested exception) an event with interruption type software interrupt, privileged software exception, or software exception, the current guest RIP is incremented by the VM-entry instruction length before being pushed on the stack.
 - If VM entry encounters an exception while injecting an event and that exception does not cause a VM exit, the current guest RIP is pushed on the stack regardless of event type or VM-entry instruction length. If the encountered exception does cause a VM exit that saves RIP, the saved RIP is current guest RIP.
- If the deliver-error-code bit (bit 11) is set in the VM-entry interruption-information field, the contents of the VM-entry exception error-code field is pushed on the stack as an error code would be pushed during delivery of an exception.
- DR6, DR7, and the IA32_DEBUGCTL MSR are not modified by event injection, even if the event has vector 1 (normal deliveries of debug exceptions, which have vector 1, do update these registers).
- If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode (RFLAGS.VM = 1), no general-protection exception can occur due to RFLAGS.IOPL < 3. A VM monitor should check RFLAGS.IOPL before injecting such an event and, if desired, inject a general-protection exception instead of a software interrupt.
- If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode with virtual-8086 mode extensions (RFLAGS.VM = CR4.VME = 1), event delivery is subject to VME-based interrupt redirection based on the software interrupt redirection bitmap in the task-state segment (TSS) as follows:
 - If bit n in the bitmap is clear (where n is the number of the software interrupt), the interrupt is directed to an 8086 program interrupt handler: the processor uses a 16-bit interrupt-vector table (IVT) located at linear address zero. If the value of RFLAGS.IOPL is less than 3, the following modifications are made to the value of RFLAGS that is pushed on the stack: IOPL is set to 3, and IF is set to the value of VIF.

1. Hardware exceptions with the following unused vectors are considered benign: 15 and 21–31. A hardware exception with vector 20 is considered benign unless the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control; in that case, it has the same severity as page faults.

2. While these items refer to RIP, the width of the value pushed (16 bits, 32 bits, or 64 bits) is determined normally.

- If bit n in the bitmap is set (where n is the number of the software interrupt), the interrupt is directed to a protected-mode interrupt handler. (In other words, the injection is treated as described in the next item.) In this case, the software interrupt does not invoke such a handler if `RFLAGS.IOPL < 3` (a general-protection exception occurs instead). However, as noted above, `RFLAGS.IOPL` cannot cause an injected software interrupt to cause such an exception. Thus, in this case, the injection invokes a protected-mode interrupt handler independent of the value of `RFLAGS.IOPL`.

Injection of events of other types are not subject to this redirection.

- If VM entry is injecting a software interrupt (not redirected as described above) or software exception, privilege checking is performed on the IDT descriptor being accessed as would be the case for executions of `INT n` , `INT3`, or `INTO` (the descriptor's DPL cannot be less than CPL). There is no checking of `RFLAGS.IOPL`, even if the guest will be in virtual-8086 mode. Failure of this check may lead to a nested exception. Injection of an event with interruption type external interrupt, NMI, hardware exception, and privileged software exception, or with interruption type software interrupt and being redirected as described above, do not perform these checks.
- If VM entry is injecting a non-maskable interrupt (NMI) and the “virtual NMIs” VM-execution control is 1, virtual-NMI blocking is in effect after VM entry.
- The transition causes a last-branch record to be logged if the LBR bit is set in the `IA32_DEBUGCTL` MSR. This is true even for events such as debug exceptions, which normally clear the LBR bit before delivery.
- The last-exception record MSRs (LERs) may be updated based on the setting of the LBR bit in the `IA32_DEBUGCTL` MSR. Events such as debug exceptions, which normally clear the LBR bit before they are delivered, and therefore do not normally update the LERs, may do so as part of VM-entry event injection.
- If injection of an event encounters a nested exception, the value of the EXT bit (bit 0) in any error code for that nested exception is determined as follows:
 - If event being injected has interruption type external interrupt, NMI, hardware exception, or privileged software exception and encounters a nested exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
 - If event being injected is a software interrupt or a software exception and encounters a nested exception, the error code for that exception clears the EXT bit.
 - If event delivery encounters a nested exception and delivery of that exception encounters another exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
 - If a double fault is produced, the error code for the double fault is 0000H (the EXT bit is clear).

26.6.1.2 VM Exits During Event Injection

An event being injected never causes a VM exit directly regardless of the settings of the VM-execution controls. For example, setting the “NMI exiting” VM-execution control to 1 does not cause a VM exit due to injection of an NMI.

However, the event-delivery process may lead to a VM exit:

- If the vector in the VM-entry interruption-information field identifies a task gate in the IDT, the attempted task switch may cause a VM exit just as it would had the injected event occurred during normal execution in VMX non-root operation (see Section 25.4.2).
- If event delivery encounters a nested exception, a VM exit may occur depending on the contents of the exception bitmap (see Section 25.2).
- If event delivery generates a double-fault exception (due to a nested exception); the logical processor encounters another nested exception while attempting to call the double-fault handler; and that exception does not cause a VM exit due to the exception bitmap; then a VM exit occurs due to triple fault (see Section 25.2).
- If event delivery injects a double-fault exception and encounters a nested exception that does not cause a VM exit due to the exception bitmap, then a VM exit occurs due to triple fault (see Section 25.2).
- If the “virtualize APIC accesses” VM-execution control is 1 and event delivery generates an access to the APIC-access page, that access is treated as described in Section 29.4 and may cause a VM exit.¹

1. “Virtualize APIC accesses” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “virtualize APIC accesses” VM-execution control were 0. See Section 24.6.2.

If the event-delivery process does cause a VM exit, the processor state before the VM exit is determined just as it would be had the injected event occurred during normal execution in VMX non-root operation. If the injected event directly accesses a task gate that cause a VM exit or if the first nested exception encountered causes a VM exit, information about the injected event is saved in the IDT-vectoring information field (see Section 27.2.4).

26.6.1.3 Event Injection for VM Entries to Real-Address Mode

If VM entry is loading CR0.PE with 0, any injected vectored event is delivered as would normally be done in real-address mode.¹ Specifically, VM entry uses the vector provided in the VM-entry interruption-information field to select a 4-byte entry from an interrupt-vector table at the linear address in IDTR.base. Further details are provided in Section 15.1.4 in Volume 3A of the *IA-32 Intel® Architecture Software Developer's Manual*.

Because bit 11 (deliver error code) in the VM-entry interruption-information field must be 0 if CR0.PE will be 0 after VM entry (see Section 26.2.1.3), vectored events injected with CR0.PE = 0 do not push an error code on the stack. This is consistent with event delivery in real-address mode.

If event delivery encounters a fault (due to a violation of IDTR.limit or of SS.limit), the fault is treated as if it had occurred during event delivery in VMX non-root operation. Such a fault may lead to a VM exit as discussed in Section 26.6.1.2.

26.6.2 Injection of Pending MTF VM Exits

If the interruption type in the VM-entry interruption-information field is 7 (other event) and the vector field is 0, VM entry causes an MTF VM exit to be pending on the instruction boundary following VM entry. This is the case even if the “monitor trap flag” VM-execution control is 0. See Section 25.5.2 for the treatment of pending MTF VM exits.

26.7 SPECIAL FEATURES OF VM ENTRY

This section details a variety of features of VM entry. It uses the following terminology: a VM entry is **vectoring** if the valid bit (bit 31) of the VM-entry interruption information field is 1 and the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception).

26.7.1 Interruptibility State

The interruptibility-state field in the guest-state area (see Table 24-3) contains bits that control blocking by STI, blocking by MOV SS, and blocking by NMI. This field impacts event blocking after VM entry as follows:

- If the VM entry is vectoring, there is no blocking by STI or by MOV SS following the VM entry, regardless of the contents of the interruptibility-state field.
- If the VM entry is not vectoring, the following apply:
 - Events are blocked by STI if and only if bit 0 in the interruptibility-state field is 1. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry; see Section 26.7.3).
 - Events are blocked by MOV SS if and only if bit 1 in the interruptibility-state field is 1. This may affect the treatment of pending debug exceptions; see Section 26.7.3. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry).
- The blocking of non-maskable interrupts (NMIs) is determined as follows:

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, VM entry must be loading CR0.PE with 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

- If the “virtual NMIs” VM-execution control is 0, NMIs are blocked if and only if bit 3 (blocking by NMI) in the interruptibility-state field is 1. If the “NMI exiting” VM-execution control is 0, execution of the IRET instruction removes this blocking (even if the instruction generates a fault). If the “NMI exiting” control is 1, IRET does not affect this blocking.
- The following items describe the use of bit 3 (blocking by NMI) in the interruptibility-state field if the “virtual NMIs” VM-execution control is 1:
 - The bit’s value does not affect the blocking of NMIs after VM entry. NMIs are not blocked in VMX non-root operation (except for ordinary blocking for other reasons, such as by the MOV SS instruction, the wait-for-SIPI state, etc.)
 - The bit’s value determines whether there is virtual-NMI blocking after VM entry. If the bit is 1, virtual-NMI blocking is in effect after VM entry. If the bit is 0, there is no virtual-NMI blocking after VM entry unless the VM entry is injecting an NMI (see Section 26.6.1.1). Execution of IRET removes virtual-NMI blocking (even if the instruction generates a fault).

If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” control must be 0; see Section 26.2.1.1.

- Blocking of system-management interrupts (SMIs) is determined as follows:
 - If the VM entry was not executed in system-management mode (SMM), SMI blocking is unchanged by VM entry.
 - If the VM entry was executed in SMM, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.

26.7.2 Activity State

The activity-state field in the guest-state area controls whether, after VM entry, the logical processor is active or in one of the inactive states identified in Section 24.4.2. The use of this field is determined as follows:

- If the VM entry is vectoring, the logical processor is in the active state after VM entry. While the consistency checks described in Section 26.3.1.5 on the activity-state field do apply in this case, the contents of the activity-state field do not determine the activity state after VM entry.
- If the VM entry is not vectoring, the logical processor ends VM entry in the activity state specified in the guest-state area. If VM entry ends with the logical processor in an inactive activity state, the VM entry generates any special bus cycle that is normally generated when that activity state is entered from the active state. If VM entry would end with the logical processor in the shutdown state and the logical processor is in SMX operation,¹ an Intel® TXT shutdown condition occurs. The error code used is 0000H, indicating “legacy shutdown.” See *Intel® Trusted Execution Technology Preliminary Architecture Specification*.
- Some activity states unconditionally block certain events. The following blocking is in effect after any VM entry that puts the processor in the indicated state:
 - The active state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the active state and in VMX non-root operation are discarded and do not cause VM exits.
 - The HLT state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the HLT state and in VMX non-root operation are discarded and do not cause VM exits.
 - The shutdown state blocks external interrupts and SIPIs. External interrupts that arrive while a logical processor is in the shutdown state and in VMX non-root operation do not cause VM exits even if the “external-interrupt exiting” VM-execution control is 1. SIPIs that arrive while a logical processor is in the shutdown state and in VMX non-root operation are discarded and do not cause VM exits.
 - The wait-for-SIPI state blocks external interrupts, non-maskable interrupts (NMIs), INIT signals, and system-management interrupts (SMIs). Such events do not cause VM exits if they arrive while a logical processor is in the wait-for-SIPI state and in VMX non-root operation.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

26.7.3 Delivery of Pending Debug Exceptions after VM Entry

The pending debug exceptions field in the guest-state area indicates whether there are debug exceptions that have not yet been delivered (see Section 24.4.2). This section describes how these are treated on VM entry.

There are no pending debug exceptions after VM entry if any of the following are true:

- The VM entry is vectoring with one of the following interruption types: external interrupt, non-maskable interrupt (NMI), hardware exception, or privileged software exception.
- The interruptibility-state field does not indicate blocking by MOV SS and the VM entry is vectoring with either of the following interruption type: software interrupt or software exception.
- The VM entry is not vectoring and the activity-state field indicates either shutdown or wait-for-SIPI.

If none of the above hold, the pending debug exceptions field specifies the debug exceptions that are pending for the guest. There are **valid pending debug exceptions** if either the BS bit (bit 14) or the enable-breakpoint bit (bit 12) is 1. If there are valid pending debug exceptions, they are handled as follows:

- If the VM entry is not vectoring, the pending debug exceptions are treated as they would had they been encountered normally in guest execution:
 - If the logical processor is not blocking such exceptions (the interruptibility-state field indicates no blocking by MOV SS), a debug exception is delivered after VM entry (see below).
 - If the logical processor is blocking such exceptions (due to blocking by MOV SS), the pending debug exceptions are held pending or lost as would normally be the case.
- If the VM entry is vectoring (with interruption type software interrupt or software exception and with blocking by MOV SS), the following items apply:
 - For injection of a software interrupt or of a software exception with vector 3 (#BP) or vector 4 (#OF) — or a privileged software exception with vector 1 (#DB) — the pending debug exceptions are treated as they would had they been encountered normally in guest execution if the corresponding instruction (INT1, INT3, or INTO) were executed after a MOV SS that encountered a debug trap.
 - For injection of a software exception with a vector other than 3 and 4, the pending debug exceptions may be lost or they may be delivered after injection (see below).

If there are no valid pending debug exceptions (as defined above), no pending debug exceptions are delivered after VM entry.

If a pending debug exception is delivered after VM entry, it has the priority of “traps on the previous instruction” (see Section 6.9 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Thus, INIT signals and system-management interrupts (SMIs) take priority of such an exception, as do VM exits induced by the TPR threshold (see Section 26.7.7) and pending MTF VM exits (see Section 26.7.8). The exception takes priority over any pending non-maskable interrupt (NMI) or external interrupt and also over VM exits due to the 1-settings of the “interrupt-window exiting” and “NMI-window exiting” VM-execution controls.

A pending debug exception delivered after VM entry causes a VM exit if the bit 1 (#DB) is 1 in the exception bitmap. If it does not cause a VM exit, it updates DR6 normally.

26.7.4 VMX-Preemption Timer

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

It is possible for the VMX-preemption timer to expire during VM entry (e.g., if the value in the VMX-preemption timer-value field is zero). If this happens (and if the VM entry was not to the wait-for-SIPI state), a VM exit occurs with its normal priority after any event injection and before execution of any instruction following VM entry. For example, any pending debug exceptions established by VM entry (see Section 26.7.3) take priority over a timer-induced VM exit. (The timer-induced VM exit will occur after delivery of the debug exception, unless that exception or its delivery causes a different VM exit.)

See Section 25.5.1 for details of the operation of the VMX-preemption timer in VMX non-root operation, including the blocking and priority of the VM exits that it causes.

26.7.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery

If “interrupt-window exiting” VM-execution control is 1, an open interrupt window may cause a VM exit immediately after VM entry (see Section 25.2 for details). If the “interrupt-window exiting” VM-execution control is 0 but the “virtual-interrupt delivery” VM-execution control is 1, a virtual interrupt may be delivered immediately after VM entry (see Section 26.3.2.5 and Section 29.2.1).

The following items detail the treatment of these events:

- These events occur after any event injection specified for VM entry.
- Non-maskable interrupts (NMIs) and higher priority events take priority over these events. These events take priority over external interrupts and lower priority events.
- These events wake the logical processor if it just entered the HLT state because of a VM entry (see Section 26.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

26.7.6 NMI-Window Exiting

The “NMI-window exiting” VM-execution control may cause a VM exit to occur immediately after VM entry (see Section 25.2 for details).

The following items detail the treatment of these VM exits:

- These VM exits follow event injection if such injection is specified for VM entry.
- Debug-trap exceptions (see Section 26.7.3) 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.
- VM exits caused by this control wake the logical processor if it just entered either the HLT state or the shutdown state because of a VM entry (see Section 26.7.2). They do not occur if the logical processor just entered the wait-for-SIPI state.

26.7.7 VM Exits Induced by the TPR Threshold

If the “use TPR shadow” and “virtualize APIC accesses” VM-execution controls are both 1 and the “virtual-interrupt delivery” VM-execution control is 0, a VM exit occurs immediately after VM entry if the value of bits 3:0 of the TPR threshold VM-execution control field is greater than the value of bits 7:4 of VTPR (see Section 29.1.1).¹

The following items detail the treatment of these VM exits:

- The VM exits are not blocked if RFLAGS.IF = 0 or by the setting of bits in the interruptibility-state field in guest-state area.
- The VM exits follow event injection if such injection is specified for VM entry.
- VM exits caused by this control take priority over system-management interrupts (SMIs), INIT signals, and lower priority events. They thus have priority over the VM exits described in Section 26.7.5, Section 26.7.6, and Section 26.7.8, as well as any interrupts or debug exceptions that may be pending at the time of VM entry.
- These VM exits wake the logical processor if it just entered the HLT state as part of a VM entry (see Section 26.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state. If such a VM exit is suppressed because the processor just entered the shutdown state, it occurs after the delivery of any event that cause the logical processor to leave the shutdown state while remaining in VMX non-root operation (e.g., due to an NMI that occurs while the “NMI-exiting” VM-execution control is 0).
- The basic exit reason is “TPR below threshold.”

1. “Virtualize APIC accesses” and “virtual-interrupt delivery” are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if these controls were 0. See Section 24.6.2.

26.7.8 Pending MTF VM Exits

As noted in Section 26.6.2, VM entry may cause an MTF VM exit to be pending immediately after VM entry. The following items detail the treatment of these VM exits:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these VM exits. These VM exits take priority over debug-trap exceptions and lower priority events.
- These VM exits wake the logical processor if it just entered the HLT state because of a VM entry (see Section 26.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

26.7.9 VM Entries and Advanced Debugging Features

VM entries are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

26.8 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE

VM-entry failures due to the checks identified in Section 26.3.1 and failures during the MSR loading identified in Section 26.4 are treated differently from those that occur earlier in VM entry. In these cases, the following steps take place:

1. Information about the VM-entry failure is recorded in the VM-exit information fields:
 - Exit reason.
 - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM-entry failure. The following numbers are used:
 33. VM-entry failure due to invalid guest state. A VM entry failed one of the checks identified in Section 26.3.1.
 34. VM-entry failure due to MSR loading. A VM entry failed in an attempt to load MSRs (see Section 26.4).
 41. VM-entry failure due to machine-check event. A machine-check event occurred during VM entry (see Section 26.9).
 - Bit 31 is set to 1 to indicate a VM-entry failure.
 - The remainder of the field (bits 30:16) is cleared.
 - Exit qualification. This field is set based on the exit reason.
 - VM-entry failure due to invalid guest state. In most cases, the exit qualification is cleared to 0. The following non-zero values are used in the cases indicated:
 1. Not used.
 2. Failure was due to a problem loading the PDPTes (see Section 26.3.1.6).
 3. Failure was due to an attempt to inject a non-maskable interrupt (NMI) into a guest that is blocking events through the STI blocking bit in the interruptibility-state field.
 4. Failure was due to an invalid VMCS link pointer (see Section 26.3.1.5).

VM-entry checks on guest-state fields may be performed in any order. Thus, an indication by exit qualification of one cause does not imply that there are not also other errors. Different processors may give different exit qualifications for the same VMCS.
 - VM-entry failure due to MSR loading. The exit qualification is loaded to indicate which entry in the VM-entry MSR-load area caused the problem (1 for the first entry, 2 for the second, etc.).
 - All other VM-exit information fields are unmodified.
2. Processor state is loaded as would be done on a VM exit (see Section 27.5). If this results in $[CR4.PAE \ \& \ CR0.PG \ \& \ \sim IA32_EFER.LMA] = 1$, page-directory-pointer-table entries (PDPTes) may be checked and loaded (see Section 27.5.4).

3. The state of blocking by NMI is what it was before VM entry.
4. MSRs are loaded as specified in the VM-exit MSR-load area (see Section 27.6).

Although this process resembles that of a VM exit, many steps taken during a VM exit do not occur for these VM-entry failures:

- Most VM-exit information fields are not updated (see step 1 above).
- The valid bit in the VM-entry interruption-information field is not cleared.
- The guest-state area is not modified.
- No MSRs are saved into the VM-exit MSR-store area.

26.9 MACHINE-CHECK EVENTS DURING VM ENTRY

If a machine-check event occurs during a VM entry, one of the following occurs:

- The machine-check event is handled as if it occurred before the VM entry:
 - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:¹
 - If the logical processor is in SMX operation, an Intel[®] TXT shutdown condition occurs. The error code used is 000CH, indicating “unrecoverable machine-check condition.”
 - If the logical processor is outside SMX operation, it goes to the shutdown state.
 - If CR4.MCE = 1, a machine-check exception (#MC) is delivered through the IDT.
- The machine-check event is handled after VM entry completes:
 - If the VM entry ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
 - If the logical processor is in SMX operation, an Intel[®] TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
 - If the logical processor is outside SMX operation, it goes to the shutdown state.
 - If the VM entry ends with CR4.MCE = 1, a machine-check exception (#MC) is generated:
 - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
 - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- A VM-entry failure occurs as described in Section 26.8. The basic exit reason is 41, for “VM-entry failure due to machine-check event.”

The first option is not used if the machine-check event occurs after any guest state has been loaded. The second option is used only if VM entry is able to load all guest state.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel[®] 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

29. Updates to Chapter 27, Volume 3C

Change bars show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

Changes to chapter: Added details on protection keys for supervisor-mode pages where applicable. Typo corrections as needed.

VM exits occur in response to certain instructions and events in VMX non-root operation as detailed in Section 25.1 through Section 25.2. VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified as described in Section 27.2.
2. Processor state is saved in the guest-state area (Section 27.3).
3. MSRs may be saved in the VM-exit MSR-store area (Section 27.4). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.
4. The following may be performed in parallel and in any order (Section 27.5):
 - Processor state is loaded based in part on the host-state area and some VM-exit controls. This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM. See Section 34.15.6 for information on how processor state is loaded by such VM exits.
 - Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 27.6). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.

VM exits are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

Section 27.1 clarifies the nature of the architectural state before a VM exit begins. The steps described above are detailed in Section 27.2 through Section 27.6.

Section 34.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, ordinary transitions to SMM are replaced by VM exits to a separate SMM monitor. Called **SMM VM exits**, these are caused by the arrival of an SMI or the execution of VMCALL in VMX root operation. SMM VM exits differ from other VM exits in ways that are detailed in Section 34.15.2.

27.1 ARCHITECTURAL STATE BEFORE A VM EXIT

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT. Note the following:

- An exception causes a VM exit **directly** if the bit corresponding to that exception is set in the exception bitmap. A non-maskable interrupt (NMI) causes a VM exit directly if the “NMI exiting” VM-execution control is 1. An external interrupt causes a VM exit directly if the “external-interrupt exiting” VM-execution control is 1. A start-up IPI (SIPI) that arrives while a logical processor is in the wait-for-SIPI activity state causes a VM exit directly. INIT signals that arrive while the processor is not in the wait-for-SIPI activity state cause VM exits directly.
- An exception, NMI, external interrupt, or software interrupt causes a VM exit **indirectly** if it does not do so directly but delivery of the event causes a nested exception, double fault, task switch, APIC access (see Section 29.4), EPT violation, EPT misconfiguration, page-modification log-full event (see Section 28.2.6), or SPP-related event (see Section 28.2.4) that causes a VM exit.
- An event **results** in a VM exit if it causes a VM exit (directly or indirectly).

The following bullets detail when architectural state is and is not updated in response to VM exits:

- If an event causes a VM exit directly, it does not update architectural state as it would have if it had it not caused the VM exit:
 - A debug exception does not update DR6, DR7, or IA32_DEBUGCTL. (Information about the nature of the debug exception is saved in the exit qualification field.)
 - A page fault does not update CR2. (The linear address causing the page fault is saved in the exit-qualification field.)

- An NMI causes subsequent NMIs to be blocked, but only after the VM exit completes.
 - An external interrupt does not acknowledge the interrupt controller and the interrupt remains pending, unless the “acknowledge interrupt on exit” VM-exit control is 1. In such a case, the interrupt controller is acknowledged and the interrupt is no longer pending.
 - The flags L0 – L3 in DR7 (bit 0, bit 2, bit 4, and bit 6) are not cleared when a task switch causes a VM exit.
 - If a task switch causes a VM exit, none of the following are modified by the task switch: old task-state segment (TSS); new TSS; old TSS descriptor; new TSS descriptor; RFLAGS.NT¹; or the TR register.
 - No last-exception record is made if the event that would do so directly causes a VM exit.
 - If a machine-check exception causes a VM exit directly, this does not prevent machine-check MSR from being updated. These are updated by the machine-check event itself and not the resulting machine-check exception.
 - If the logical processor is in an inactive state (see Section 24.4.2) and not executing instructions, some events may be blocked but others may return the logical processor to the active state. Unblocked events may cause VM exits.² If an unblocked event causes a VM exit directly, a return to the active state occurs only after the VM exit completes.³ The VM exit generates any special bus cycle that is normally generated when the active state is entered from that activity state.
- MTF VM exits (see Section 25.5.2 and Section 26.7.8) are not blocked in the HLT activity state. If an MTF VM exit occurs in the HLT activity state, the logical processor returns to the active state only after the VM exit completes. MTF VM exits are blocked the shutdown state and the wait-for-SIPI state.
- If an event causes a VM exit indirectly, the event does update architectural state:
 - A debug exception updates DR6, DR7, and the IA32_DEBUGCTL MSR. No debug exceptions are considered pending.
 - A page fault updates CR2.
 - An NMI causes subsequent NMIs to be blocked before the VM exit commences.
 - An external interrupt acknowledges the interrupt controller and the interrupt is no longer pending.
 - If the logical processor had been in an inactive state, it enters the active state and, before the VM exit commences, generates any special bus cycle that is normally generated when the active state is entered from that activity state.
 - There is no blocking by STI or by MOV SS when the VM exit commences.
 - Processor state that is normally updated as part of delivery through the IDT (CS, RIP, SS, RSP, RFLAGS) is not modified. However, the incomplete delivery of the event may write to the stack.
 - The treatment of last-exception records is implementation dependent:
 - Some processors make a last-exception record when beginning the delivery of an event through the IDT (before it can encounter a nested exception). Such processors perform this update even if the event encounters a nested exception that causes a VM exit (including the case where nested exceptions lead to a triple fault).
 - Other processors delay making a last-exception record until event delivery has reached some event handler successfully (perhaps after one or more nested exceptions). Such processors do not update the last-exception record if a VM exit or triple fault occurs before an event handler is reached.

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

2. If a VM exit takes the processor from an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.

3. An exception is made if the logical processor had been inactive due to execution of MWAIT; in this case, it is considered to have become active before the VM exit.

- If the “virtual NMIs” VM-execution control is 1, VM entry injects an NMI, and delivery of the NMI causes a nested exception, double fault, task switch, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event, or APIC access that causes a VM exit, virtual-NMI blocking is in effect before the VM exit commences.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the “NMI exiting” VM-execution control is 0, any blocking by NMI is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.3.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the “virtual NMIs” VM-execution control is 1, virtual-NMI blocking is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.3.
- Suppose that a VM exit is caused directly by an x87 FPU Floating-Point Error (#MF) or by any of the following events if the event was unblocked due to (and given priority over) an x87 FPU Floating-Point Error: an INIT signal, an external interrupt, an NMI, an SMI; or a machine-check exception. In these cases, there is no blocking by STI or by MOV SS when the VM exit commences.
- Normally, a last-branch record may be made when an event is delivered through the IDT. However, if such an event results in a VM exit before delivery is complete, no last-branch record is made.
- If machine-check exception results in a VM exit, processor state is suspect and may result in suspect state being saved to the guest-state area. A VM monitor should consult the RIPV and EIPV bits in the IA32_MCG_STATUS MSR before resuming a guest that caused a VM exit resulting from a machine-check exception.
- If a VM exit results from a fault, APIC access (see Section 29.4), EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered while executing an instruction, data breakpoints due to that instruction may have been recognized and information about them may be saved in the pending debug exceptions field (unless the VM exit clears that field; see Section 27.3.4).
- The following VM exits are considered to happen after an instruction is executed:
 - VM exits resulting from debug traps (single-step, I/O breakpoints, and data breakpoints).
 - VM exits resulting from debug exceptions (data breakpoints) whose recognition was delayed by blocking by MOV SS.
 - VM exits resulting from some machine-check exceptions.
 - Trap-like VM exits due to execution of MOV to CR8 when the “CR8-load exiting” VM-execution control is 0 and the “use TPR shadow” VM-execution control is 1 (see Section 29.3). (Such VM exits can occur only from 64-bit mode and thus only on processors that support Intel 64 architecture.)
 - Trap-like VM exits due to execution of WRMSR when the “use MSR bitmaps” VM-execution control is 1; the value of ECX is in the range 800H–8FFH; and the bit corresponding to the ECX value in write bitmap for low MSRs is 0; and the “virtualize x2APIC mode” VM-execution control is 1. See Section 29.5.
 - VM exits caused by APIC-write emulation (see Section 29.4.3.2) that result from APIC accesses as part of instruction execution.

For these VM exits, the instruction’s modifications to architectural state complete before the VM exit occurs. Such modifications include those to the logical processor’s interruptibility state (see Table 24-3). If there had been blocking by MOV SS, POP SS, or STI before the instruction executed, such blocking is no longer in effect.

A VM exit that occurs in enclave mode sets bit 27 of the exit-reason field and bit 4 of the guest interruptibility-state field. Before such a VM exit is delivered, an Asynchronous Enclave Exit (AEX) occurs (see Chapter 39, “Enclave Exiting Events”). An AEX modifies architectural state (Section 39.3). In particular, the processor establishes the following architectural state as indicated:

- The following bits in RFLAGS are cleared: CF, PF, AF, ZF, SF, OF, and RF.
- FS and GS are restored to the values they had prior to the most recent enclave entry.
- RIP is loaded with the AEP of interrupted enclave thread.
- RSP is loaded from the URSP field in the enclave’s state-save area (SSA).

27.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS

VM exits begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields. Section 27.2.1 to Section 27.2.5 detail the use of these fields.

In addition to updating the VM-exit information fields, the valid bit (bit 31) is cleared in the VM-entry interruption-information field. If bit 5 of the IA32_VMX_MISC MSR (index 485H) is read as 1 (see Appendix A.6), the value of IA32_EFER.LMA is stored into the “IA-32e mode guest” VM-entry control.¹

27.2.1 Basic VM-Exit Information

Section 24.9.1 defines the basic VM-exit information fields. The following items detail their use.

- **Exit reason.**
 - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM exit. Appendix C lists the numbers used and their meaning.
 - Bit 27 of this field is set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits include those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode. A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1).
 - The remainder of the field (bits 31:28 and bits 26:16) is cleared to 0 (certain SMM VM exits may set some of these bits; see Section 34.15.2.3).²
- **Exit qualification.** This field is saved for VM exits due to the following causes: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); system-management interrupts (SMIs) that arrive immediately after the execution of I/O instructions; task switches; INVEPT; INVLPG; INVPCID; INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; MWAIT; accesses to the APIC-access page (see Section 29.4); EPT violations (see Section 28.2.3.2); EOI virtualization (see Section 29.1.4); APIC-write emulation (see Section 29.4.3.3); page-modification log full (see Section 28.2.6); and SPP-related events (see Section 28.2.4). For all other VM exits, this field is cleared. The following items provide details:
 - For a debug exception, the exit qualification contains information about the debug exception. The information has the format given in Table 27-1.

Table 27-1. Exit Qualification for Debug Exceptions

Bit Position(s)	Contents
3:0	B3 - B0. When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if its corresponding enabling bit in DR7 is not set.
12:4	Not currently defined.
13	BD. When set, this bit indicates that the cause of the debug exception is “debug register access detected.”
14	BS. When set, this bit indicates that the cause of the debug exception is either the execution of a single instruction (if RFLAGS.TF = 1 and IA32_DEBUGCTL.BTF = 0) or a taken branch (if RFLAGS.TF = DEBUGCTL.BTF = 1).
15	Not currently defined.

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
 2. Bit 31 of this field is set on certain VM-entry failures; see Section 26.8.

Table 27-1. Exit Qualification for Debug Exceptions (Contd.)

Bit Position(s)	Contents
16	RTM. When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i>). ¹
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 16 to indicate an RTM-related exception, while this field **sets** the bit to indicate that condition.

- For a page-fault exception, the exit qualification contains the linear address that caused the page fault. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.
- For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.
- For a task switch, the exit qualification contains details about the task switch, encoded as shown in Table 27-2.
- For INVLPG, the exit qualification contains the linear-address operand of the instruction.
 - On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
 - If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

Table 27-2. Exit Qualification for Task Switches

Bit Position(s)	Contents
15:0	Selector of task-state segment (TSS) to which the guest attempted to switch
29:16	Not currently defined
31:30	Source of task switch initiation: 0: CALL instruction 1: IRET instruction 2: JMP instruction 3: Task gate in IDT
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

- For INVEPT, INVPCID, INVVPID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, and XSAVES, the exit qualification receives the value of the instruction's displacement field, which is sign-extended to 64 bits if necessary (32 bits on processors that do not support Intel 64 architecture). If the instruction has no displacement (for example, has a register operand), zero is stored into the exit qualification.

On processors that support Intel 64 architecture, an exception is made for RIP-relative addressing (used only in 64-bit mode). Such addressing causes an instruction to use an address that is the sum of the displacement field and the value of RIP that references the following instruction. In this case, the exit qualification is loaded with the sum of the displacement field and the appropriate RIP value.

In all cases, bits of this field beyond the instruction’s address size are undefined. For example, suppose that the address-size field in the VM-exit instruction-information field (see Section 24.9.4 and Section 27.2.5) reports an *n*-bit address size. Then bits 63:*n* (bits 31:*n* on processors that do not support Intel 64 architecture) of the instruction displacement are undefined.

- For a control-register access, the exit qualification contains information about the access and has the format given in Table 27-3.
- For MOV DR, the exit qualification contains information about the instruction and has the format given in Table 27-4.
- For an I/O instruction, the exit qualification contains information about the instruction and has the format given in Table 27-5.
- For MWAIT, the exit qualification contains a value that indicates whether address-range monitoring hardware was armed. The exit qualification is set either to 0 (if address-range monitoring hardware is not armed) or to 1 (if address-range monitoring hardware is armed).
- For an APIC-access VM exit resulting from a linear access or a guest-physical access to the APIC-access page (see Section 29.4), the exit qualification contains information about the access and has the format given in Table 27-6.¹

If the access to the APIC-access page occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

Such a VM exit that set bits 15:12 of the exit qualification to 0000b (data read during instruction execution) or 0001b (data write during instruction execution) set bit 12—which distinguishes data read from data write—to that which would have been stored in bit 1—W/R—of the page-fault error code had the access caused a page fault instead of an APIC-access VM exit. This implies the following:

- For an APIC-access VM exit caused by the CLFLUSH and CLFLUSHOPT instructions, the access type is “data read during instruction execution.”
- For an APIC-access VM exit caused by the ENTER instruction, the access type is “data write during instruction execution.”

Table 27-3. Exit Qualification for Control-Register Accesses

Bit Positions	Contents
3:0	Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.
5:4	Access type: 0 = MOV to CR 1 = MOV from CR 2 = CLTS 3 = LMSW
6	LMSW operand type: 0 = register 1 = memory For CLTS and MOV CR, cleared to 0
7	Not currently defined

1. The exit qualification is undefined if the access was part of the logging of a branch record or a processor-event-based-sampling (PEBS) record to the DS save area. It is recommended that software configure the paging structures so that no address in the DS save area translates to an address on the APIC-access page.

Table 27-3. Exit Qualification for Control-Register Accesses (Contd.)

Bit Positions	Contents
11:8	For MOV CR, the general-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) For CLTS and LMSW, cleared to 0
15:12	Not currently defined
31:16	For LMSW, the LMSW source data For CLTS and MOV CR, cleared to 0
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

- For an APIC-access VM exit caused by the MASKMOVQ instruction or the MASKMOVDQU instruction, the access type is "data write during instruction execution."
- For an APIC-access VM exit caused by the MONITOR instruction, the access type is "data read during instruction execution."

Such a VM exit stores 1 for bit 31 for IDT-vectoring information field (see Section 27.2.4) if and only if it sets bits 15:12 of the exit qualification to 0011b (linear access during event delivery) or 1010b (guest-physical access during event delivery).

See Section 29.4.4 for further discussion of these instructions and APIC-access VM exits.

For APIC-access VM exits resulting from physical accesses to the APIC-access page (see Section 29.4.6), the exit qualification is undefined.

- For an EPT violation, the exit qualification contains information about the access causing the EPT violation and has the format given in Table 27-7.

As noted in that table, the format and meaning of the exit qualification depends on the setting of the "mode-based execute control for EPT" VM-execution control and whether the processor supports advanced VM-exit information for EPT violations.¹

An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations.

Table 27-4. Exit Qualification for MOV DR

Bit Position(s)	Contents
2:0	Number of debug register
3	Not currently defined
4	Direction of access (0 = MOV to DR; 1 = MOV from DR)

1. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

Table 27-4. Exit Qualification for MOV DR (Contd.)

Bit Position(s)	Contents
7:5	Not currently defined
11:8	General-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8 - 15 = R8 - R15, respectively
63:12	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

Table 27-5. Exit Qualification for I/O Instructions

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte Other values not used
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)
15:7	Not currently defined
31:16	Port number (as specified in DX or in an immediate operand)
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

Table 27-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses

Bit Position(s)	Contents
11:0	<ul style="list-style-type: none"> ▪ If the APIC-access VM exit is due to a linear access, the offset of access within the APIC page. ▪ Undefined if the APIC-access VM exit is due a guest-physical access

Table 27-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses (Contd.)

Bit Position(s)	Contents
15:12	Access type: 0 = linear access for a data read during instruction execution 1 = linear access for a data write during instruction execution 2 = linear access for an instruction fetch 3 = linear access (read or write) during event delivery 10 = guest-physical access during event delivery 15 = guest-physical access for an instruction fetch or during instruction execution Other values not used
16	If the APIC-access VM exit is due to a guest-physical access, this bit is set if the access was asynchronous to instruction execution and not part of event delivery. (The bit is set if the access is related to trace output by Intel PT; see Section 25.5.4.) Otherwise, this bit is cleared.
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

Bit 12 reports “NMI unblocking due to IRET”; see Section 27.2.3.

Bit 16 is set if the VM exit occurs during trace-address pre-translation (TAPT); see Section 25.5.4.

- For VM exits caused as part of EOI virtualization (Section 29.1.4), bits 7:0 of the exit qualification are set to vector of the virtual interrupt that was dismissed by the EOI virtualization. Bits above bit 7 are cleared.
- For APIC-write VM exits (Section 29.4.3.3), bits 11:0 of the exit qualification are set to the page offset of the write access that caused the VM exit.¹ Bits above bit 11 are cleared.
- For a VM exit due to a page-modification log-full event (Section 28.2.6), bit 12 of the exit qualification reports “NMI unblocking due to IRET.” Bit 16 is set if the VM exit occurs during TAPT. All other bits of the exit qualification are undefined.
- For a VM exit due to an SPP-related event (Section 28.2.4), bit 11 of the exit qualification indicates the type of event: 0 indicates an SPP misconfiguration and 1 indicates an SPP miss. Bit 12 of the exit qualification reports “NMI unblocking due to IRET.” Bit 16 is set if the VM exit occurs during TAPT. All other bits of the exit qualification are undefined.
- **Guest linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:
 - VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
 - VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

Table 27-7. Exit Qualification for EPT Violations

Bit Position(s)	Contents
0	Set if the access causing the EPT violation was a data read. ¹
1	Set if the access causing the EPT violation was a data write. ¹
2	Set if the access causing the EPT violation was an instruction fetch.

1. Execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit; the exit qualification for such an APIC-write VM exit is 3FOH.

Table 27-7. Exit Qualification for EPT Violations (Contd.)

Bit Position(s)	Contents
3	The logical-AND of bit 0 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was readable). ²
4	The logical-AND of bit 1 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was writeable).
5	The logical-AND of bit 2 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. If the “mode-based execute control for EPT” VM-execution control is 0, this indicates whether the guest-physical address was executable. If that control is 1, this indicates whether the guest-physical address was executable for supervisor-mode linear addresses.
6	If the “mode-based execute control” VM-execution control is 0, the value of this bit is undefined. If that control is 1, this bit is the logical-AND of bit 10 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. In this case, it indicates whether the guest-physical address was executable for user-mode linear addresses.
7	Set if the guest linear-address field is valid. The guest linear-address field is valid for all EPT violations except those resulting from an attempt to load the guest PDPTes as part of the execution of the MOV CR instruction and those due to trace-address pre-translation (TAPT; Section 25.5.4).
8	If bit 7 is 1: <ul style="list-style-type: none"> ▪ Set if the access causing the EPT violation is to a guest-physical address that is the translation of a linear address. ▪ Clear if the access causing the EPT violation is to a paging-structure entry as part of a page walk or the update of an accessed or dirty bit. Reserved if bit 7 is 0 (cleared to 0).
9	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, ³ this bit is 0 if the linear address is a supervisor-mode linear address and 1 if it is a user-mode linear address. (If CRO.PG = 0, the translation of every linear address is a user-mode linear address and thus this bit will be 1.) Otherwise, this bit is undefined.
10	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, ³ this bit is 0 if paging translates the linear address to a read-only page and 1 if it translates to a read/write page. (If CRO.PG = 0, every linear address is read/write and thus this bit will be 1.) Otherwise, this bit is undefined.
11	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, ³ this bit is 0 if paging translates the linear address to an executable page and 1 if it translates to an execute-disable page. (If CRO.PG = 0, CR4.PAE = 0, or IA32_EFER.NXE = 0, every linear address is executable and thus this bit will be 0.) Otherwise, this bit is undefined.
12	NMI unblocking due to IRET (see Section 27.2.3).
13	Set if the access causing the EPT violation was a shadow-stack access.
14	If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), this bit is the same as bit 60 in the EPT paging-structure entry that maps the page of the guest-physical address of the access causing the EPT violation. Otherwise (or if translation of the guest-physical address terminates before reaching an EPT paging-structure entry that maps a page), this bit is undefined.
15	Not currently defined.
16	This bit is set if the access was asynchronous to instruction execution not the result of event delivery. (The bit is set if the access is related to trace output by Intel PT; see Section 25.5.4.) Otherwise, this bit is cleared.
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. If accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations (see Section 28.2.3.2). If such an access causes an EPT violation, the processor sets both bit 0 and bit 1 of the exit qualification.
2. Bits 5:3 are cleared to 0 if any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present (see Section 28.2.2).
3. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

- VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 27-7; these are all EPT violations except those resulting from an attempt to load the PDPTes as of execution of the MOV CR instruction and those due to TAPT). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the EPT violation occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

- VM exits due to SPP-related events.
 - For all other VM exits, the field is undefined.
- **Guest-physical address.** For a VM exit due to an EPT violation, an EPT misconfiguration, or an SPP-related event, this field receives the guest-physical address that caused the EPT violation or EPT misconfiguration. For all other VM exits, the field is undefined.

If the EPT violation or EPT misconfiguration occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

27.2.2 Information for VM Exits Due to Vectored Events

Section 24.9.2 defines fields containing information for VM exits due to the following events: exceptions (including those generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs).¹ Such VM exits include those that occur on an attempt at a task switch that causes an exception before generating the VM exit due to the task switch that causes the VM exit.

The following items detail the use of these fields:

- **VM-exit interruption information** (format given in Table 24-16). The following items detail how this field is established for VM exits due to these events:
 - For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the vector.
 - Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 5 (privileged software exception), or 6 (software exception). Hardware exceptions comprise all exceptions except the following:
 - Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
 - Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for *n*.

- Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).¹ If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).
- Bit 12 reports “NMI unblocking due to IRET”; see Section 27.2.3. The value of this bit is undefined if the VM exit is due to a double fault (the interruption type is hardware exception and the vector is 8).
- Bits 30:13 are always set to 0.
- Bit 31 is always set to 1.

For other VM exits (including those due to external interrupts when the “acknowledge interrupt on exit” VM-exit control is 0), the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- VM-exit interruption error code.
 - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the VM-exit interruption-information field, this field receives the error code that would have been pushed on the stack had the event causing the VM exit been delivered normally through the IDT. The EXT bit is set in this field exactly when it would be set normally. For exceptions that occur during the delivery of double fault (if the IDT-vectoring information field indicates a double fault), the EXT bit is set to 1, assuming that (1) that the exception would produce an error code normally (if not incident to double-fault delivery) and (2) that the error code uses the EXT bit (not for page faults, which use a different format).
 - For other VM exits, the value of this field is undefined.

27.2.3 Information About NMI Unblocking Due to IRET

A VM exit may occur during execution of the IRET instruction for reasons including the following: faults, EPT violations, page-modification log-full events, or SPP-related events.

An execution of IRET that commences while non-maskable interrupts (NMIs) are blocked will unblock NMIs even if a fault or VM exit occurs; the state saved by such a VM exit will indicate that NMIs were not blocked.

VM exits for the reasons enumerated above provide more information to software by saving a bit called “NMI unblocking due to IRET.” This bit is defined if (1) either the “NMI exiting” VM-execution control is 0 or the “virtual NMIs” VM-execution control is 1; (2) the VM exit does not set the valid bit in the IDT-vectoring information field (see Section 27.2.4); and (3) the VM exit is not due to a double fault. In these cases, the bit is defined as follows:

- The bit is 1 if the VM exit resulted from a memory access as part of execution of the IRET instruction and one of the following holds:
 - The “virtual NMIs” VM-execution control is 0 and blocking by NMI (see Table 24-3) was in effect before execution of IRET.
 - The “virtual NMIs” VM-execution control is 1 and virtual-NMI blocking was in effect before execution of IRET.
- The bit is 0 for all other relevant VM exits.

For VM exits due to faults, NMI unblocking due to IRET is saved in bit 12 of the VM-exit interruption-information field (Section 27.2.2). For VM exits due to EPT violations, page-modification log-full events, and SPP-related events, NMI unblocking due to IRET is saved in bit 12 of the exit qualification (Section 27.2.1).

(Executions of IRET may also incur VM exits due to APIC accesses and EPT misconfigurations. These VM exits do not report information about NMI unblocking due to IRET.)

1. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

27.2.4 Information for VM Exits During Event Delivery

Section 24.9.3 defined fields containing information for VM exits that occur while delivering an event through the IDT and as a result of any of the following cases:¹

- A fault occurs during event delivery and causes a VM exit (because the bit associated with the fault is set to 1 in the exception bitmap).
- A task switch is invoked through a task gate in the IDT. The VM exit occurs due to the task switch only after the initial checks of the task switch pass (see Section 25.4.2).
- Event delivery causes an APIC-access VM exit (see Section 29.4).
- An EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that occurs during event delivery.

These fields are used for VM exits that occur during delivery of events injected as part of VM entry (see Section 26.6.1.2).

A VM exit is not considered to occur during event delivery in any of the following circumstances:

- The original event causes the VM exit directly (for example, because the original event is a non-maskable interrupt (NMI) and the “NMI exiting” VM-execution control is 1).
- The original event results in a double-fault exception that causes the VM exit directly.
- The VM exit occurred as a result of fetching the first instruction of the handler invoked by the event delivery.
- The VM exit is caused by a triple fault.

The following items detail the use of these fields:

- IDT-vectoring information (format given in Table 24-17). The following items detail how this field is established for VM exits that occur during event delivery:
 - If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the vector.
 - Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).

Hardware exceptions comprise all exceptions except the following:²

- Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
- Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

- Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).³ If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).
- Bit 12 is undefined.

1. This includes the case in which a VM exit occurs while delivering a software interrupt (INT *n*) through the 16-bit IVT (interrupt vector table) that is used in virtual-8086 mode with virtual-machine extensions (if RFLAGS.VM = CR4.VME = 1).

2. In the following items, INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for *n*.

3. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

- Bits 30:13 are always set to 0.
- Bit 31 is always set to 1.

For other VM exits, the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- IDT-vectoring error code.
 - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the IDT-vectoring information field, this field receives the error code that would have been pushed on the stack by the event that was being delivered through the IDT at the time of the VM exit. The EXT bit is set in this field when it would be set normally.
 - For other VM exits, the value of this field is undefined.

27.2.5 Information for VM Exits Due to Instruction Execution

Section 24.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

- **VM-exit instruction length.** This field is used in the following cases:
 - For fault-like VM exits due to attempts to execute one of the following instructions that cause VM exits unconditionally (see Section 25.1.2) or based on the settings of VM-execution controls (see Section 25.1.3): CLTS, CPUID, ENCLS, GETSEC, HLT, IN, INS, INVLD, INVEPT, INVLPG, INVPCID, INVVPID, LGDT, LIDT, LLDT, LMSW, LTR, MONITOR, MOV CR, MOV DR, MWAIT, OUT, OUTS, PAUSE, RDMSR, RDPIC, RDRAND, RDSEED, RDTSC, RDTSCP, RSM, SGDT, SIDT, SLDT, STR, TPAUSE, UMWAIT, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, WBINVD, WRMSR, XRSTORS, XSETBV, and XSAVES.¹
 - For VM exits due to software exceptions (those generated by executions of INT3 or INTO) or privileged software exceptions (those generated by executions of INT1).
 - For VM exits due to faults encountered during delivery of a software interrupt, privileged software exception, or software exception.
 - For VM exits due to attempts to effect a task switch via instruction execution. These are VM exits that produce an exit reason indicating task switch and either of the following:
 - An exit qualification indicating execution of CALL, IRET, or JMP instruction.
 - An exit qualification indicating a task gate in the IDT and an IDT-vectoring information field indicating that the task gate was encountered during delivery of a software interrupt, privileged software exception, or software exception.
 - For APIC-access VM exits and for VM exits caused by EPT violations, page-modification log-full events, and SPP-related events encountered during delivery of a software interrupt, privileged software exception, or software exception.²
 - For VM exits due to executions of VMFUNC that fail because one of the following is true:
 - EAX indicates a VM function that is not enabled (the bit at position EAX is 0 in the VM-function controls; see Section 25.5.6.2).
 - EAX = 0 and either ECX ≥ 512 or the value of ECX selects an invalid tentative EPTP value (see Section 25.5.6.3).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

-
1. This item applies only to fault-like VM exits. It does not apply to trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1 or to those following executions of the WRMSR instruction when the “virtualize x2APIC mode” VM-execution control is 1.
 2. The VM-exit instruction-length field is not defined following APIC-access VM exits resulting from physical accesses (see Section 29.4.6) even if encountered during delivery of a software interrupt, privileged software exception, or software exception.

The cases of VM exits encountered during delivery of a software interrupt, privileged software exception, or software exception include those encountered during delivery of events injected as part of VM entry (see Section 26.6.1.2). If the original event was injected as part of VM entry, this field receives the value of the VM-entry instruction length.

All VM exits other than those listed in the above items leave this field undefined.

If the VM exit occurred in enclave mode, this field is cleared (none of the previous items apply).

Table 27-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS

Bit Position(s)	Content
6:0	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
14:10	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for VM exits due to execution of INS.
31:18	Undefined.

- VM-exit instruction information.** For VM exits due to attempts to execute INS, INVEPT, INVPCID, INVVPID, LIDT, LGDT, LLDT, LTR, OUTS, RDRAND, RDSEED, SIDT, SGDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, or XSAVES, this field receives information about the instruction that caused the VM exit. The format of the field depends on the identity of the instruction causing the VM exit:
 - For VM exits due to attempts to execute INS or OUTS, the field has the format is given in Table 27-8.¹
 - For VM exits due to attempts to execute INVEPT, INVPCID, or INVVPID, the field has the format is given in Table 27-9.
 - For VM exits due to attempts to execute LIDT, LGDT, SIDT, or SGDT, the field has the format is given in Table 27-10.
 - For VM exits due to attempts to execute LLDT, LTR, SLDT, or STR, the field has the format is given in Table 27-11.
 - For VM exits due to attempts to execute RDRAND, RDSEED, TPAUSE, or UMWAIT, the field has the format is given in Table 27-12.
 - For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, or XSAVES, the field has the format is given in Table 27-13.
 - For VM exits due to attempts to execute VMREAD or VMWRITE, the field has the format is given in Table 27-14.

For all other VM exits, the field is undefined, unless the VM exit occurred in enclave mode, in which case the field is cleared.

1. The format of the field was undefined for these VM exits on the first processors to support the virtual-machine extensions. Software can determine whether the format specified in Table 27-8 is used by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

- **I/O RCX, I/O RSI, I/O RDI, I/O RIP.** These fields are undefined except for SMM VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions. See Section 34.15.2.3. Note that, if the VM exit occurred in enclave mode, these fields are all cleared.

Table 27-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for memory instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Reg2 (same encoding as IndexReg above)

Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
11	Operand size: 0: 16-bit 1: 32-bit Undefined for VM exits from 64-bit mode.
14:12	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
29:28	Instruction identity: 0: SGDT 1: SIDT 2: LGDT 3: LIDT

Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT (Contd.)

Bit Position(s)	Content
31:30	Undefined.

Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).

Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR (Contd.)

Bit Position(s)	Content
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
29:28	Instruction identity: 0: SLDT 1: STR 2: LLDT 3: LTR
31:30	Undefined.

Table 27-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND, RDSEED, TPAUSE, and UMWAIT

Bit Position(s)	Content
2:0	Undefined.
6:3	Operand register (destination for RDRAND and RDSEED; source for TPAUSE and UMWAIT): 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture)
10:7	Undefined.
12:11	Operand size: 0: 16-bit 1: 32-bit 2: 64-bit The value 3 is not used.
31:13	Undefined.

Table 27-13. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.

Table 27-13. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES (Contd.)

Bit Position(s)	Content
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Undefined.

Table 27-14. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.

Table 27-14. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE (Contd.)

Bit Position(s)	Content
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
31:28	Reg2 (same encoding as Reg1 above)

27.3 SAVING GUEST STATE

VM exits save certain components of processor state into corresponding fields in the guest-state area of the VMCS (see Section 24.4). On processors that support Intel 64 architecture, the full value of each natural-width field (see Section 24.11.2) is saved regardless of the mode of the logical processor before and after the VM exit.

In general, the state saved is that which was in the logical processor at the time the VM exit commences. See Section 27.1 for a discussion of which architectural updates occur at that time.

Section 27.3.1 through Section 27.3.4 provide details for how various components of processor state are saved. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

27.3.1 Saving Control Registers, Debug Registers, and MSRs

Contents of certain control registers, debug registers, and MSRs is saved as follows:

- The contents of CR0, CR3, CR4, and the IA32_SYSENTER_CS, IA32_SYSENTER_ESP, and IA32_SYSENTER_EIP MSRs are saved into the corresponding fields. Bits 63:32 of the IA32_SYSENTER_CS MSR are not saved. On processors that do not support Intel 64 architecture, bits 63:32 of the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are not saved.
- If the “save debug controls” VM-exit control is 1, the contents of DR7 and the IA32_DEBUGCTL MSR are saved into the corresponding fields. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always saved data into these fields.
- If the “save IA32_PAT” VM-exit control is 1, the contents of the IA32_PAT MSR are saved into the corresponding field.
- If the “save IA32_EFER” VM-exit control is 1, the contents of the IA32_EFER MSR are saved into the corresponding field.
- If the processor supports either the 1-setting of the “load IA32_BNDCFGS” VM-entry control or that of the “clear IA32_BNDCFGS” VM-exit control, the contents of the IA32_BNDCFGS MSR are saved into the corresponding field.
- If the processor supports either the 1-setting of the “load IA32_RTIT_CTL” VM-entry control or that of the “clear IA32_RTIT_CTL” VM-exit control, the contents of the IA32_RTIT_CTL MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the “load CET” VM-entry control, the contents of the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are saved into the corresponding fields. On processors that do not support Intel 64 architecture, bits 63:32 of these MSRs are not saved.
- If the processor supports the 1-setting of the “load PKRS” VM-entry control, the contents of the IA32_PKRS MSR are saved into the corresponding field.
- The value of the SMBASE field is undefined after all VM exits except SMM VM exits. See Section 34.15.2.

27.3.2 Saving Segment Registers and Descriptor-Table Registers

For each segment register (CS, SS, DS, ES, FS, GS, LDTR, or TR), the values saved for the base-address, segment-limit, and access rights are based on whether the register was unusable (see Section 24.4.1) before the VM exit:

- If the register was unusable, the values saved into the following fields are undefined: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in the access-rights field. The following exceptions apply:
 - CS.
 - The base-address and segment-limit fields are saved.
 - The L, D, and G bits are saved in the access-rights field.
 - SS.
 - DPL is saved in the access-rights field.
 - On processors that support Intel 64 architecture, bits 63:32 of the value saved for the base address are always zero.
 - DS and ES. On processors that support Intel 64 architecture, bits 63:32 of the values saved for the base addresses are always zero.
 - FS and GS. The base-address field is saved.
 - LDTR. The value saved for the base address is always canonical.
- If the register was not unusable, the values saved into the following fields are those which were in the register before the VM exit: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in access rights.
- Bits 31:17 and 11:8 in the access-rights field are always cleared. Bit 16 is set to 1 if and only if the segment is unusable.

The contents of the GDTR and IDTR registers are saved into the corresponding base-address and limit fields.

27.3.3 Saving RIP, RSP, RFLAGS, and SSP

The contents of the RIP, RSP, RFLAGS, and SSP (shadow-stack pointer) registers are saved as follows:

- The value saved in the RIP field is determined by the nature and cause of the VM exit:
 - If the VM exit occurred in enclave mode, the value saved is the AEP of interrupted enclave thread (the remaining items do not apply).
 - If the VM exit occurs due to by an attempt to execute an instruction that causes VM exits unconditionally or that has been configured to cause a VM exit via the VM-execution controls, the value saved references that instruction.
 - If the VM exit is caused by an occurrence of an INIT signal, a start-up IPI (SIPI), or system-management interrupt (SMI), the value saved is that which was in RIP before the event occurred.
 - If the VM exit occurs due to the 1-setting of either the “interrupt-window exiting” VM-execution control or the “NMI-window exiting” VM-execution control, the value saved is that which would be in the register had the VM exit not occurred.
 - If the VM exit is due to an external interrupt, non-maskable interrupt (NMI), or hardware exception (as defined in Section 27.2.2), the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate,¹ or into the old task-state segment had the event been delivered through a task gate).
 - If the VM exit is due to a triple fault, the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate, or into the old task-state segment had the event been delivered through a task gate) had delivery of the double fault not encountered the nested exception that caused the triple fault.
 - If the VM exit is due to a software exception (due to an execution of INT3 or INTO) or a privileged software exception (due to an execution of INT1), the value saved references the INT3, INTO, or INT1 instruction that caused that exception.
 - Suppose that the VM exit is due to a task switch that was caused by execution of CALL, IRET, or JMP or by execution of a software interrupt (INT *n*), software exception (due to execution of INT3 or INTO), or privileged software exception (due to execution of INT1) that encountered a task gate in the IDT. The value saved references the instruction that caused the task switch (CALL, IRET, JMP, INT *n*, INT3, INTO, INT1).
 - Suppose that the VM exit is due to a task switch that was caused by a task gate in the IDT that was encountered for any reason except the direct access by a software interrupt or software exception. The value saved is that which would have been saved in the old task-state segment had the task switch completed normally.
 - If the VM exit is due to an execution of MOV to CR8 or WRMSR that reduced the value of bits 7:4 of VTPR (see Section 29.1.1) below that of TPR threshold VM-execution control field (see Section 29.1.2), the value saved references the instruction following the MOV to CR8 or WRMSR.
 - If the VM exit was caused by APIC-write emulation (see Section 29.4.3.2) that results from an APIC access as part of instruction execution, the value saved references the instruction following the one whose execution caused the APIC-write emulation.
- The contents of the RSP register are saved into the RSP field.
- With the exception of the resume flag (RF; bit 16), the contents of the RFLAGS register is saved into the RFLAGS field. RFLAGS.RF is saved as follows:
 - If the VM exit occurred in enclave mode, the value saved is 0 (the remaining items do not apply).
 - If the VM exit is caused directly by an event that would normally be delivered through the IDT, the value saved is that which would appear in the saved RFLAGS image (either that which would be saved on the stack had the event been delivered through a trap or interrupt gate² or into the old task-state segment had

1. The reference here is to the full value of RIP before any truncation that would occur had the stack width been only 32 bits or 16 bits.

2. The reference here is to the full value of RFLAGS before any truncation that would occur had the stack width been only 32 bits or 16 bits.

the event been delivered through a task gate) had the event been delivered through the IDT. See below for VM exits due to task switches caused by task gates in the IDT.

- If the VM exit is caused by a triple fault, the value saved is that which the logical processor would have in RF in the RFLAGS register had the triple fault taken the logical processor to the shutdown state.
- If the VM exit is caused by a task switch (including one caused by a task gate in the IDT), the value saved is that which would have been saved in the RFLAGS image in the old task-state segment (TSS) had the task switch completed normally without exception.
- If the VM exit is caused by an attempt to execute an instruction that unconditionally causes VM exits or one that was configured to do with a VM-execution control, the value saved is 0.¹
- For APIC-access VM exits and for VM exits caused by EPT violations, EPT misconfigurations, page-modification log-full events, or SPP-related events, the value saved depends on whether the VM exit occurred during delivery of an event through the IDT:
 - If the VM exit stored 0 for bit 31 for IDT-vectoring information field (because the VM exit did not occur during delivery of an event through the IDT; see Section 27.2.4), the value saved is 1.
 - If the VM exit stored 1 for bit 31 for IDT-vectoring information field (because the VM exit did occur during delivery of an event through the IDT), the value saved is the value that would have appeared in the saved RFLAGS image had the event been delivered through the IDT (see above).
- For all other VM exits, the value saved is the value RFLAGS.RF had before the VM exit occurred.
- If the processor supports the 1-setting of the “load CET” VM-entry control, the contents of the SSP register are saved into the SSP field.

27.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

- The activity-state field is saved with the logical processor’s activity state before the VM exit.² See Section 27.1 for details of how events leading to a VM exit may affect the activity state.
- The interruptibility-state field is saved to reflect the logical processor’s interruptibility before the VM exit.
 - See Section 27.1 for details of how events leading to a VM exit may affect this state.
 - VM exits that end outside system-management mode (SMM) save bit 2 (blocking by SMI) as 0 regardless of the state of such blocking before the VM exit.
 - Bit 3 (blocking by NMI) is treated specially if the “virtual NMIs” VM-execution control is 1. In this case, the value saved for this field does not indicate the blocking of NMIs but rather the state of virtual-NMI blocking.
 - Bit 4 (enclave interruption) is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.
- The pending debug exceptions field is saved as clear for all VM exits except the following:
 - A VM exit caused by an INIT signal, a machine-check exception, or a system-management interrupt (SMI).

1. This is true even if RFLAGS.RF was 1 before the instruction was executed. If, in response to such a VM exit, a VM monitor re-enters the guest to re-execute the instruction that caused the VM exit (for example, after clearing the VM-execution control that caused the VM exit), the instruction may encounter a code breakpoint that has already been processed. A VM monitor can avoid this by setting the guest value of RFLAGS.RF to 1 before resuming guest software.

2. If this activity state was an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.

- A VM exit with basic exit reason “TPR below threshold”,¹ “virtualized EOI”, “APIC write”, or “monitor trap flag.”
- A VM exit due to trace-address pre-translation (TAPT; see Section 25.5.4). Such VM exits can have basic exit reason “APIC access,” “EPT violation,” “EPT misconfiguration,” “page-modification log full,” or “SPP-related event.” When due to TAPT, these VM exits (with the exception of those due to EPT misconfigurations) set bit 16 of the exit qualification, indicating that they are asynchronous to instruction execution and not part of event delivery.
- VM exits that are not caused by debug exceptions and that occur while there is MOV-SS blocking of debug exceptions.

For VM exits that do not clear the field, the value saved is determined as follows:

- Each of bits 3:0 may be set if it corresponds to a matched breakpoint. This may be true even if the corresponding breakpoint is not enabled in DR7.
- Suppose that a VM exit is due to an INIT signal, a machine-check exception, or an SMI; or that a VM exit has basic exit reason “TPR below threshold” or “monitor trap flag.” In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit.

If the VM exit occurs immediately after VM entry, the value saved may match that which was loaded on VM entry (see Section 26.7.3). Otherwise, the following items apply:

- Bit 12 (enabled breakpoint) is set to 1 in any of the following cases:
 - If there was at least one matched data or I/O breakpoint that was enabled in DR7.
 - If it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.7.3) and the VM exit occurred before those exceptions were either delivered or lost.
 - If the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). (This does not apply to VM exits with basic exit reason “monitor trap flag.”)

In other cases, bit 12 is cleared to 0.

- Bit 14 (BS) is set if RFLAGS.TF = 1 in either of the following cases:
 - IA32_DEBUGCTL.BTF = 0 and the cause of a pending debug exception was the execution of a single instruction.
 - IA32_DEBUGCTL.BTF = 1 and the cause of a pending debug exception was a taken branch.
- Bit 16 (RTM) is set if a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions had been enabled. (This does not apply to VM exits with basic exit reason “monitor trap flag.”)
- Suppose that a VM exit is due to another reason (but not a debug exception) and occurs while there is MOV-SS blocking of debug exceptions. In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit. If the VM exit occurs immediately after VM entry (no instructions were executed in VMX non-root operation), the value saved may match that which was loaded on VM entry (see Section 26.7.3). Otherwise, the following items apply:
 - Bit 12 (enabled breakpoint) is set to 1 if there was at least one matched data or I/O breakpoint that was enabled in DR7. Bit 12 is also set if it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.7.3) and the VM exit occurred before those exceptions were either delivered or lost. In other cases, bit 12 is cleared to 0.
 - The setting of bit 14 (BS) is implementation-specific. However, it is not set if RFLAGS.TF = 0 or IA32_DEBUGCTL.BTF = 1.
- The reserved bits in the field are cleared.
- If the “save VMX-preemption timer value” VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 25.5.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value

1. This item includes VM exits that occur as a result of certain VM entries (Section 26.7.7).

0 if the timer expired during VM exit. (If the “save VMX-preemption timer value” VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)

- If the logical processor supports the 1-setting of the “enable EPT” VM-execution control, values are saved into the four (4) PDPTE fields as follows:
 - If the “enable EPT” VM-execution control is 1 and the logical processor was using PAE paging at the time of the VM exit, the PDPTE values currently in use are saved:¹
 - The values saved into bits 11:9 of each of the fields is undefined.
 - If the value saved into one of the fields has bit 0 (present) clear, the value saved into bits 63:1 of that field is undefined. That value need not correspond to the value that was loaded by VM entry or to any value that might have been loaded in VMX non-root operation.
 - If the value saved into one of the fields has bit 0 (present) set, the value saved into bits 63:12 of the field is a guest-physical address.
 - If the “enable EPT” VM-execution control is 0 or the logical processor was not using PAE paging at the time of the VM exit, the values saved are undefined.

27.4 SAVING MSRS

After processor state is saved to the guest-state area, values of MSRs may be stored into the VM-exit MSR-store area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-store count) is processed in order by storing the value of the MSR indexed by bits 31:0 (as they would be read by RDMSR) into bits 127:64. Processing of an entry fails in either of the following cases:

- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be read only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMBASE is an MSR that can be read only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be saved on VM exits for model-specific reasons. A processor may prevent certain MSRs (based on the value of bits 31:0) from being stored on VM exits, even if they can normally be read by RDMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.
- Bits 63:32 of the entry are not all 0.
- An attempt to read the MSR indexed by bits 31:0 would cause a general-protection exception if executed via RDMSR with CPL = 0.

A VMX abort occurs if processing fails for any entry. See Section 27.7.

27.5 LOADING HOST STATE

Processor state is updated on VM exits in the following ways:

- Some state is loaded from or otherwise determined by the contents of the host-state area.
- Some state is determined by VM-exit controls.
- Some state is established in the same way on every VM exit.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order.

1. A logical processor uses PAE paging if CRO.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*. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM exit functions as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.

On processors that support Intel 64 architecture, the full values of each 64-bit field loaded (for example, the base address for GDTR) is loaded regardless of the mode of the logical processor before and after the VM exit.

The loading of host state is detailed in Section 27.5.1 to Section 27.5.5. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

A logical processor is in IA-32e mode after a VM exit only if the “host address-space size” VM-exit control is 1. If the logical processor was in IA-32e mode before the VM exit and this control is 0, a VMX abort occurs. See Section 27.7.

In addition to loading host state, VM exits clear address-range monitoring (Section 27.5.6).

After the state loading described in this section, VM exits may load MSRs from the VM-exit MSR-load area (see Section 27.6). This loading occurs only after the state loading described in this section.

27.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
 - The following bits are not modified:
 - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 23.8).¹
 - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width (they are cleared to 0).² (This item applies only to processors that support Intel 64 architecture.)
 - For CR4, any bits that are fixed in VMX operation (see Section 23.8).
 - CR4.PAE is set to 1 if the “host address-space size” VM-exit control is 1.
 - CR4.PCIDE is set to 0 if the “host address-space size” VM-exit control is 0.
- DR7 is set to 400H.
- The following MSRs are established as follows:
 - The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.
 - The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
 - The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP fields, respectively.

If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

If the processor supports the Intel 64 architecture with $N < 64$ linear-address bits, each of bits 63:N is set to the value of bit $N-1$.³
 - The following steps are performed on processors that support Intel 64 architecture:
 - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 27.5.2).
 - The LMA and LME bits in the IA32_EFER MSR are each loaded with the setting of the “host address-space size” VM-exit control.

1. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.

2. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

3. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- If the “load IA32_PERF_GLOBAL_CTRL” VM-exit control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.
- If the “load IA32_PAT” VM-exit control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.
- If the “load IA32_EFER” VM-exit control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.
- If the “clear IA32_BNDCFGS” VM-exit control is 1, the IA32_BNDCFGS MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
- If the “clear IA32_RTIT_CTL” VM-exit control is 1, the IA32_RTIT_CTL MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
- If the “load CET” VM-exit control is 1, the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are loaded from the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR fields, respectively.
If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
If the processor supports the Intel 64 architecture with $N < 64$ linear-address bits, each of bits 63:N is set to the value of bit N-1.
- If the “load PKRS” VM-exit control is 1, the IA32_PKRS MSR is loaded from the IA32_PKRS field. Bits 63:32 of that MSR are maintained with zeroes.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 27.6.

27.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified Section 26.3.1.2 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).
- The base address is set as follows:
 - CS. Cleared to zero.
 - SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.
 - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.
If the processor supports the Intel 64 architecture and the processor supports $N < 64$ linear-address bits, each of bits 63:N is set to the value of bit N-1.¹ The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
 - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports $N < 64$ linear-address bits, each of bits 63:N is set to the value of bit N-1.
- The segment limit is set as follows:
 - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFFFFH and a G-bit setting of 1).
 - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
 - TR. Set to 00000067H.
- The type field and S bit are set as follows:

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
- SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
- TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).
- The DPL is set as follows:
 - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.
 - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.
- The P bit is set as follows:
 - CS, TR. Set to 1.
 - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the “host address-space size” VM-exit control. Because the value of this control is also loaded into IA32_EFER.LMA (see Section 27.5.1), no VM exit is ever to compatibility mode (which requires IA32_EFER.LMA = 1 and CS.L = 0).
- D/B.
 - CS. Loaded with the inverse of the setting of the “host address-space size” VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
 - SS. Set to 1.
 - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
 - TR. Set to 0.
- G.
 - CS. Set to 1.
 - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
 - TR. Set to 0.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined (although the base address is always canonical).

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports $N < 64$ linear-address bits, each of bits 63:N of each base address is set to the value of bit N-1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

27.5.3 Loading Host RIP, RSP, RFLAGS, and SSP

RIP and RSP are loaded from the RIP field and the RSP field, respectively. RFLAGS is cleared, except bit 1, which is always set.

If the “load CET” VM-exit control is 1, SSP (shadow-stack pointer) is loaded from the SSP field.

27.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LMA = 0, the logical processor uses **PAE paging**. See Section 4.4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.¹ When in PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs and, if they are valid, loads them into the processor (into internal, non-architectural registers).

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

A VM exit is to a VMM that uses PAE paging if (1) bit 5 (corresponding to CR4.PAE) is set in the CR4 field in the host-state area of the VMCS; and (2) the “host address-space size” VM-exit control is 0. Such a VM exit may check the validity of the PDPTes referenced by the CR3 field in the host-state area of the VMCS. Such a VM exit must check their validity if either (1) PAE paging was not in use before the VM exit; or (2) the value of CR3 is changing as a result of the VM exit. A VM exit to a VMM that does not use PAE paging must not check the validity of the PDPTes.

A VM exit that checks the validity of the PDPTes uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the PDPTes that would be loaded (e.g., because a reserved bit is set), a VMX abort occurs (see Section 27.7). If a VM exit to a VMM that uses PAE does not cause a VMX abort, the PDPTes are loaded into the processor as would MOV to CR3, using the value of CR3 being load by the VM exit.

27.5.5 Updating Non-Register State

VM exits affect the non-register state of a logical processor as follows:

- A logical processor is always in the active state after a VM exit.
- Event blocking is affected as follows:
 - There is no blocking by STI or by MOV SS after a VM exit.
 - VM exits caused directly by non-maskable interrupts (NMIs) cause blocking by NMI (see Table 24-3). Other VM exits do not affect blocking by NMI. (See Section 27.1 for the case in which an NMI causes a VM exit indirectly.)
- There are no pending debug exceptions after a VM exit.

Section 28.3 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM exits invalidate cached mappings:

- If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP).
- VM exits are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

27.5.6 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 8.10.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. VM exits clear any address-range monitoring that may be in effect.

27.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C000100H (the IA32_FS_BASE MSR) or C000101H (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-

specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.¹

If processing fails for any entry, a VMX abort occurs. See Section 27.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

27.7 VMX ABORTS

A problem encountered during a VM exit leads to a **VMX abort**. A VMX abort takes a logical processor into a shut-down state as described below.

A VMX abort does not modify the VMCS data in the VMCS region of any active VMCS. The contents of these data are thus suspect after the VMX abort.

On a VMX abort, a logical processor saves a nonzero 32-bit VMX-abort indicator field at byte offset 4 in the VMCS region of the VMCS whose misconfiguration caused the failure (see Section 24.2). The following values are used:

1. There was a failure in saving guest MSRs (see Section 27.4).
2. Host checking of the page-directory-pointer-table entries (PDPTes) failed (see Section 27.5.4).
3. The current VMCS has been corrupted (through writes to the corresponding VMCS region) in such a way that the logical processor cannot complete the VM exit properly.
4. There was a failure on loading host MSRs (see Section 27.6).
5. There was a machine-check event during VM exit (see Section 27.8).
6. The logical processor was in IA-32e mode before the VM exit and the “host address-space size” VM-entry control was 0 (see Section 27.5).

Some of these causes correspond to failures during the loading of state from the host-state area. Because the loading of such state may be done in any order (see Section 27.5) a VM exit that might lead to a VMX abort for multiple reasons (for example, the current VMCS may be corrupt and the host PDPTes might not be properly configured). In such cases, the VMX-abort indicator could correspond to any one of those reasons.

A logical processor never reads the VMX-abort indicator in a VMCS region and writes it only with one of the non-zero values mentioned above. The VMX-abort indicator allows software on one logical processor to diagnose the VMX-abort on another. For this reason, it is recommended that software running in VMX root operation zero the VMX-abort indicator in the VMCS region of any VMCS that it uses.

After saving the VMX-abort indicator, operation of a logical processor experiencing a VMX abort depends on whether the logical processor is in SMX operation:²

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000DH, indicating “VMX abort.” See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*.
- If the logical processor is outside SMX operation, it issues a special bus cycle (to notify the chipset) and enters the **VMX-abort shutdown state**. RESET is the only event that wakes a logical processor from the VMX-abort shutdown state. The following events do not affect a logical processor in this state: machine-check events;

1. Note the following about processors that support Intel 64 architecture. If CRO.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CRO.PG is always 1 in VMX operation, the IA32_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.

2. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

INIT signals; external interrupts; non-maskable interrupts (NMIs); start-up IPIs (SIPIs); and system-management interrupts (SMIs).

27.8 MACHINE-CHECK EVENTS DURING VM EXIT

If a machine-check event occurs during VM exit, one of the following occurs:

- The machine-check event is handled as if it occurred before the VM exit:
 - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:¹
 - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating “unrecoverable machine-check condition.”
 - If the logical processor is outside SMX operation, it goes to the shutdown state.
 - If CR4.MCE = 1, a machine-check exception (#MC) is generated:
 - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
 - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- The machine-check event is handled after VM exit completes:
 - If the VM exit ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
 - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
 - If the logical processor is outside SMX operation, it goes to the shutdown state.
 - If the VM exit ends with CR4.MCE = 1, a machine-check exception (#MC) is delivered through the host IDT.
- A VMX abort is generated (see Section 27.7). The logical processor blocks events as done normally in VMX abort. The VMX abort indicator is 5, for “machine-check event during VM exit.”

The first option is not used if the machine-check event occurs after any host state has been loaded. The second option is used only if VM entry is able to load all host state.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

30. Updates to Chapter 28, Volume 3C

Change bars show changes to Chapter 28 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

Changes to chapter: Addition to section 28.3.3 "Invalidating Cached Translation Information".

The architecture for VMX operation includes two features that support address translation: virtual-processor identifiers (VPIDs) and the extended page-table mechanism (EPT). VPIDs are a mechanism for managing translations of linear addresses. EPT defines a layer of address translation that augments the translation of linear addresses.

Section 28.1 details the architecture of VPIDs. Section 28.2 provides the details of EPT. Section 28.3 explains how a logical processor may cache information from the paging structures, how it may use that cached information, and how software can managed the cached information.

28.1 VIRTUAL PROCESSOR IDENTIFIERS (VPIDS)

The original architecture for VMX operation required VMX transitions to flush the TLBs and paging-structure caches. This ensured that translations cached for the old linear-address space would not be used after the transition.

Virtual-processor identifiers (**VPIDs**) introduce to VMX operation a facility by which a logical processor may cache information for multiple linear-address spaces. When VPIDs are used, VMX transitions may retain cached information and the logical processor switches to a different linear-address space.

Section 28.3 details the mechanisms by which a logical processor manages information cached for multiple address spaces. A logical processor may tag some cached information with a 16-bit VPID. This section specifies how the current VPID is determined at any point in time:

- The current VPID is 0000H in the following situations:
 - Outside VMX operation. (This includes operation in system-management mode under the default treatment of SMIs and SMM with VMX operation; see Section 34.14.)
 - In VMX root operation.
 - In VMX non-root operation when the “enable VPID” VM-execution control is 0.
- If the logical processor is in VMX non-root operation and the “enable VPID” VM-execution control is 1, the current VPID is the value of the VPID VM-execution control field in the VMCS. (VM entry ensures that this value is never 0000H; see Section 26.2.1.1.)

VPIDs and PCIDs (see Section 4.10.1) can be used concurrently. When this is done, the processor associates cached information with both a VPID and a PCID. Such information is used only if the current VPID and PCID **both** match those associated with the cached information.

28.2 THE EXTENDED PAGE TABLE MECHANISM (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 addresses that would normally be treated as physical addresses (and used to access memory) are instead treated as **guest-physical addresses**. Guest-physical addresses are translated by traversing a set of **EPT paging structures** to produce physical addresses that are used to access memory.

- Section 28.2.1 gives an overview of EPT.
- Section 28.2.2 describes operation of EPT-based address translation.
- Section 28.2.3 discusses VM exits that may be caused by EPT.
- Section 28.2.7 describes interactions between EPT and memory typing.

28.2.1 EPT Overview

EPT is used when the “enable EPT” VM-execution control is 1.¹ It translates the guest-physical addresses used in VMX non-root operation and those used by VM entry for event injection.

The translation from guest-physical addresses to physical addresses is determined by a set of **EPT paging structures**. The EPT paging structures are similar to those used to translate linear addresses while the processor is in IA-32e mode. Section 28.2.2 gives the details of the EPT paging structures.

If CR0.PG = 1, linear addresses are translated through paging structures referenced through control register CR3. While the “enable EPT” VM-execution control is 1, these are called **guest paging structures**. There are no guest paging structures if CR0.PG = 0.¹

When the “enable EPT” VM-execution control is 1, the identity of **guest-physical addresses** depends on the value of CR0.PG:

- If CR0.PG = 0, each linear address is treated as a guest-physical address.
- If CR0.PG = 1, guest-physical addresses are those derived from the contents of control register CR3 and the guest paging structures. (This includes the values of the PDPTes, which logical processors store in internal, non-architectural registers.) The latter includes (in page-table entries and in other paging-structure entries for which bit 7—PS—is 1) the addresses to which linear addresses are translated by the guest paging structures.

If CR0.PG = 1, the translation of a linear address to a physical address requires multiple translations of guest-physical addresses using EPT. Assume, for example, that CR4.PAE = CR4.PSE = 0. The translation of a 32-bit linear address then operates as follows:

- Bits 31:22 of the linear address select an entry in the guest page directory located at the guest-physical address in CR3. The guest-physical address of the guest page-directory entry (PDE) is translated through EPT to determine the guest PDE’s physical address.
- Bits 21:12 of the linear address select an entry in the guest page table located at the guest-physical address in the guest PDE. The guest-physical address of the guest page-table entry (PTE) is translated through EPT to determine the guest PTE’s physical address.
- Bits 11:0 of the linear address is the offset in the page frame located at the guest-physical address in the guest PTE. The guest-physical address determined by this offset is translated through EPT to determine the physical address to which the original linear address translates.

In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called **EPT violations** and cause VM exits. See Section 28.2.3.

A processor uses EPT to translate guest-physical addresses only when those addresses are used to access memory. This principle implies the following:

- The MOV to CR3 instruction loads CR3 with a guest-physical address. Whether that address is translated through EPT depends on whether PAE paging is being used.²
 - If PAE paging is not being used, the instruction does not use that address to access memory and does **not** cause it to be translated through EPT. (If CR0.PG = 1, the address will be translated through EPT on the next memory accessing using a linear address.)
 - If PAE paging is being used, the instruction loads the four (4) page-directory-pointer-table entries (PDPTes) from that address and it **does** cause the address to be translated through EPT.
- Section 4.4.1 identifies executions of MOV to CR0 and MOV to CR4 that load the PDPTes from the guest-physical address in CR3. Such executions cause that address to be translated through EPT.
- The PDPTes contain guest-physical addresses. The instructions that load the PDPTes (see above) do not use those addresses to access memory and do **not** cause them to be translated through EPT. The address in a PDPTE will be translated through EPT on the next memory accessing using a linear address that uses that PDPTE.

1. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, the logical processor operates as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. 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*.

28.2.2 EPT Translation Mechanism

The EPT translation mechanism uses only bits 47:0 of each guest-physical address.¹ It uses a page-walk length of 4, meaning that at most 4 EPT paging-structure entries are accessed to translate a guest-physical address.²

These 48 bits are partitioned by the logical processor to traverse the EPT paging structures:

- A 4-KByte naturally aligned EPT PML4 table is located at the physical address specified in bits 51:12 of the extended-page-table pointer (EPTP), a VM-execution control field (see Table 24-8 in Section 24.6.11). An EPT PML4 table comprises 512 64-bit entries (EPT PML4Es). An EPT PML4E is selected using the physical address defined as follows:
 - Bits 63:52 are all 0.
 - Bits 51:12 are from the EPTP.
 - Bits 11:3 are bits 47:39 of the guest-physical address.
 - Bits 2:0 are all 0.

Because an EPT PML4E is identified using bits 47:39 of the guest-physical address, it controls access to a 512-GByte region of the guest-physical-address space. The format of an EPT PML4E is given in Table 28-1.

Table 28-1. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 512-GByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 512-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 512-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry ¹
51:N	Reserved (must be 0)
63:52	Ignored

1. No processors supporting the Intel 64 architecture support more than 48 physical-address bits. Thus, no such processor can produce a guest-physical address with more than 48 bits. An attempt to use such an address causes a page fault. An attempt to load CR3 with such an address causes a general-protection fault. If PAE paging is being used, an attempt to load CR3 that would load a PDPTe with such an address causes a general-protection fault.

2. Future processors may include support for other EPT page-walk lengths. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine what EPT page-walk lengths are supported.

NOTES:

1. N is the physical-address width supported by the processor. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- A 4-KByte naturally aligned EPT page-directory-pointer table is located at the physical address specified in bits 51:12 of the EPT PML4E. An EPT page-directory-pointer table comprises 512 64-bit entries (EPT PDPTes). An EPT PDPTE is selected using the physical address defined as follows:
 - Bits 63:52 are all 0.
 - Bits 51:12 are from the EPT PML4E.
 - Bits 11:3 are bits 38:30 of the guest-physical address.
 - Bits 2:0 are all 0.

Because an EPT PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the EPT PDPTE depends on the value of bit 7 in that entry:¹

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page. The final physical address is computed as follows:
 - Bits 63:52 are all 0.
 - Bits 51:30 are from the EPT PDPTE.
 - Bits 29:0 are from the original guest-physical address.

The format of an EPT PDPTE that maps a 1-GByte page is given in Table 28-2.

- If bit 7 of the EPT PDPTE is 0, a 4-KByte naturally aligned EPT page directory is located at the physical address specified in bits 51:12 of the EPT PDPTE. The format of an EPT PDPTE that references an EPT page directory is given in Table 28-3.

1. Not all processors allow bit 7 of an EPT PDPTE to be set to 1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether this is allowed.

Table 28-2. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 1-GByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 1-GByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte page controlled by this entry
5:3	EPT memory type for this 1-GByte page (see Section 28.2.7)
6	Ignore PAT memory type for this 1-GByte page (see Section 28.2.7)
7	Must be 1 (otherwise, this entry references an EPT page directory)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
29:12	Reserved (must be 0)
(N-1):30	Physical address of the 1-GByte page referenced by this entry ¹
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 1-GByte page mapped by this entry (see Section 28.2.3.2). Ignored if bit 7 of EPTP is 0
62:61	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 25.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

NOTES:

1. N is the physical-address width supported by the logical processor.

Table 28-3. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that References an EPT Page Directory

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 1-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 1-GByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte region controlled by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page directory referenced by this entry ¹
51:N	Reserved (must be 0)
63:52	Ignored

NOTES:

1. N is the physical-address width supported by the logical processor.

An EPT page-directory comprises 512 64-bit entries (PDEs). An EPT PDE is selected using the physical address defined as follows:

- Bits 63:52 are all 0.
- Bits 51:12 are from the EPT PDPTE.
- Bits 11:3 are bits 29:21 of the guest-physical address.
- Bits 2:0 are all 0.

Because an EPT PDE is identified using bits 47:21 of the guest-physical address, it controls access to a 2-MByte region of the guest-physical-address space. Use of the EPT PDE depends on the value of bit 7 in that entry:

- If bit 7 of the EPT PDE is 1, the EPT PDE maps a 2-MByte page. The final physical address is computed as follows:
 - Bits 63:52 are all 0.
 - Bits 51:21 are from the EPT PDE.
 - Bits 20:0 are from the original guest-physical address.

The format of an EPT PDE that maps a 2-MByte page is given in Table 28-4.

- If bit 7 of the EPT PDE is 0, a 4-KByte naturally aligned EPT page table is located at the physical address specified in bits 51:12 of the EPT PDE. The format of an EPT PDE that references an EPT page table is given in Table 28-5.

An EPT page table comprises 512 64-bit entries (PTEs). An EPT PTE is selected using a physical address defined as follows:

- Bits 63:52 are all 0.

Table 28-4. Format of an EPT Page-Directory Entry (PDE) that Maps a 2-MByte Page

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 2-MByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 2-MByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte page controlled by this entry
5:3	EPT memory type for this 2-MByte page (see Section 28.2.7)
6	Ignore PAT memory type for this 2-MByte page (see Section 28.2.7)
7	Must be 1 (otherwise, this entry references an EPT page table)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
20:12	Reserved (must be 0)
(N-1):21	Physical address of the 2-MByte page referenced by this entry ¹
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 2-MByte page mapped by this entry (see Section 28.2.3.2). Ignored if bit 7 of EPTP is 0
62:61	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 25.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

NOTES:

1. N is the physical-address width supported by the logical processor.

- Bits 51:12 are from the EPT PDE.
- Bits 11:3 are bits 20:12 of the guest-physical address.
- Bits 2:0 are all 0.
- Because an EPT PTE is identified using bits 47:12 of the guest-physical address, every EPT PTE maps a 4-KByte page. The final physical address is computed as follows:
 - Bits 63:52 are all 0.
 - Bits 51:12 are from the EPT PTE.

- Bits 11:0 are from the original guest-physical address.

The format of an EPT PTE is given in Table 28-6.

An EPT paging-structure entry is **present** if any of bits 2:0 is 1; otherwise, the entry is **not present**. The processor ignores bits 62:3 and uses the entry neither to reference another EPT paging-structure entry nor to produce a physical address. A reference using a guest-physical address whose translation encounters an EPT paging-structure that is not present causes an EPT violation (see Section 28.2.3.2). (If the “EPT-violation #VE” VM-execution control is 1, the EPT violation is convertible to a virtualization exception only if bit 63 is 0; see Section 25.5.7.1. If the “EPT-violation #VE” VM-execution control is 0, this bit is ignored.)

Table 28-5. Format of an EPT Page-Directory Entry (PDE) that References an EPT Page Table

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 2-MByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 2-MByte region controlled by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte region controlled by this entry
6:3	Reserved (must be 0)
7	Must be 0 (otherwise, this entry maps a 2-MByte page)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte region controlled by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of 4-KByte aligned EPT page table referenced by this entry ¹
51:N	Reserved (must be 0)
63:52	Ignored

NOTES:

1. N is the physical-address width supported by the logical processor.

NOTE

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 bits 2:0 are all 0 but bit 10 is 1, the entry is used normally to reference another EPT paging-structure entry or to produce a physical address.

The discussion above describes how the EPT paging structures reference each other and how the logical processor traverses those structures when translating a guest-physical address. It does not cover all details of the translation process. Additional details are provided as follows:

- Situations in which the translation process may lead to VM exits (sometimes before the process completes) are described in Section 28.2.3.
- Interactions between the EPT translation mechanism and memory typing are described in Section 28.2.7.

Figure 28-1 gives a summary of the formats of the EPTP and the EPT paging-structure entries. For the EPT paging structure entries, it identifies separately the format of entries that map pages, those that reference other EPT

paging structures, and those that do neither because they are not present; bits 2:0 and bit 7 are highlighted because they determine how a paging-structure entry is used. (Figure 28-1 does not comprehend the fact that, if the “mode-based execute control for EPT” VM-execution control is 1, an entry is present if any of bits 2:0 or bit 10 is 1.)

28.2.3 EPT-Induced VM Exits

Accesses using guest-physical addresses may cause VM exits due to EPT misconfigurations, EPT violations, and page-modification log-full events. An **EPT misconfiguration** occurs when, in the course of translating a guest-physical address, the logical processor encounters an EPT paging-structure entry that contains an unsupported value (see Section 28.2.3.1). An **EPT violation** occurs when there is no EPT misconfiguration but the EPT paging-structure entries disallow an access using the guest-physical address (see Section 28.2.3.2). A **page-modification log-full event** occurs when the logical processor determines a need to create a page-modification log entry and the current log is full (see Section 28.2.6).

These events occur only due to an attempt to access memory with a guest-physical address. Loading CR3 with a guest-physical address with the MOV to CR3 instruction can cause neither an EPT configuration nor an EPT violation until that address is used to access a paging structure.¹

If the “EPT-violation #VE” VM-execution control is 1, certain EPT violations may cause virtualization exceptions instead of VM exits. See Section 25.5.7.1.

28.2.3.1 EPT Misconfigurations

An EPT misconfiguration occurs if translation of a guest-physical address encounters an EPT paging-structure that meets any of the following conditions:

- Bit 0 of the entry is clear (indicating that data reads are not allowed) and bit 1 is set (indicating that data writes are allowed).
- Either of the following if the processor does not support execute-only translations:
 - Bit 0 of the entry is clear (indicating that data reads are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).²
 - The “mode-based execute control for EPT” VM-execution control is 1, bit 0 of the entry is clear (indicating that data reads are not allowed), and bit 10 is set (indicating that instruction fetches are allowed from user-mode linear addresses).

Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP to determine whether execute-only translations are supported (see Appendix A.10).

- The entry is present (see Section 28.2.2) and one of the following holds:
 - A reserved bit is set. This includes the setting of a bit in the range 51:12 that is beyond the logical processor’s physical-address width.³ See Section 28.2.2 for details of which bits are reserved in which EPT paging-structure entries.
 - The entry is the last one used to translate a guest physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE) and the value of bits 5:3 (EPT memory type) is 2, 3, or 7 (these values are reserved).

EPT misconfigurations result when an EPT paging-structure entry is configured with settings reserved for future functionality. Software developers should be aware that such settings may be used in the future and that an EPT paging-structure entry that causes an EPT misconfiguration on one processor might not do so in the future.

1. If the logical processor is using PAE paging—because CR0.PG = CR4.PAE = 1 and IA32_EFER.LMA = 0—the MOV to CR3 instruction loads the PDPTes from memory using the guest-physical address being loaded into CR3. In this case, therefore, the MOV to CR3 instruction may cause an EPT misconfiguration, an EPT violation, or a page-modification log-full event.

2. If the “mode-based execute control for EPT” VM-execution control is 1, setting bit 2 indicates that instruction fetches are allowed from supervisor-mode linear addresses.

3. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

Table 28-6. Format of an EPT Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry
1	Write access; indicates whether writes are allowed to the 4-KByte page referenced by this entry
2	If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 4-KByte page controlled by this entry
5:3	EPT memory type for this 4-KByte page (see Section 28.2.7)
6	Ignore PAT memory type for this 4-KByte page (see Section 28.2.7)
7	Ignored
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 28.2.5). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 4-KByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N-1):12	Physical address of the 4-KByte page referenced by this entry ¹
51:N	Reserved (must be 0)
59:52	Ignored
60	Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 4-KByte page mapped by this entry (see Section 28.2.3.2). Ignored if bit 7 of EPTP is 0
61	Sub-page write permissions. If the “sub-page write permissions for EPT” VM-execution control is 1, writes to individual 128-byte regions of the 4-KByte page referenced by this entry may be allowed even if the page would normally not be writable (see Section 28.2.4). If “sub-page write permissions for EPT” VM-execution control is 0, this bit is ignored.
62	Ignored
63	Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 25.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

NOTES:

1. N is the physical-address width supported by the logical processor.

28.2.3.2 EPT Violations

An EPT violation may occur during an access using a guest-physical address whose translation does not cause an EPT misconfiguration. An EPT violation occurs in any of the following situations:

- Translation of the guest-physical address encounters an EPT paging-structure entry that is not present (see Section 28.2.2).

- The access is a data read and, for any byte to be read, bit 0 (read access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte. Reads by the logical processor of guest paging structures to translate a linear address are considered to be data reads.

6	6	6	5	5	5	5	5	5	5	5	M ¹	M-1	3	3	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	9	8	7	6	5	4	3	2	1	0										
3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0										
Reserved											Address of EPT PML4 table											Rsvd.	S	A	EPT	EPT	EPTP ³																										
Ignored											Rsvd.	Address of EPT page-directory-pointer table											lg	X	lg	A	Reserved	X	W	R	PML4E: present ⁶																						
S	V	Ignored																																						0	0	0	PML4E: not present										
S	V	Ign.	S	S	Ignored											Rsvd.	Physical address of 1GB page	Reserved											lg	X	U	D	A	1	P	A	EPT	MT	X	W	R	PDPTE: 1GB page											
Ignored											Rsvd.	Address of EPT page directory											lg	X	U	D	A	0	Rsvd.	X	W	R	PDPTE: page directory																				
S	V	Ignored																																						0	0	0	PDPTE: not present										
S	V	Ign.	S	S	Ignored											Rsvd.	Physical address of 2MB page	Reserved											lg	X	U	D	A	1	P	A	EPT	MT	X	W	R	PDE: 2MB page											
Ignored											Rsvd.	Address of EPT page table											lg	X	U	D	A	0	Rsvd.	X	W	R	PDE: page table																				
S	V	Ignored																																						0	0	0	PDE: not present										
S	V	lg	S	P	S	Ignored											Rsvd.	Physical address of 4KB page											lg	X	U	D	A	lg	P	A	EPT	MT	X	W	R	PTE: 4KB page											
S	V	Ignored																																						0	0	0	PTE: not present										

Figure 28-1. Formats of EPTP and EPT Paging-Structure Entries

NOTES:

- M is an abbreviation for MAXPHYADDR.
 - Supervisor shadow-stack control.
 - See Section 24.6.11 for details of the EPTP.
 - Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 0, this bit is ignored.
 - Execute access. If the "mode-based execute control for EPT" VM-execution control is 1, this bit controls execute access for supervisor-mode linear addresses.
 - 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. This table does not comprehend that fact.
 - Suppress #VE. If the "EPT-violation #VE" VM-execution control is 0, this bit is ignored.
 - Supervisor shadow-stack page. If bit 7 of the EPTP is 0, this bit is ignored.
 - Sub-page write permissions. If the "sub-page write permissions for EPT" VM-execution control is 0, this bit is ignored.
- The access is a data write and, for any byte to be written, bit 1 (write access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte. Writes by the logical processor to guest paging structures to update accessed and dirty flags are considered to be data writes.

If bit 6 of the EPT pointer (EPTP) is 1 (enabling accessed and dirty flags for EPT), processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations. Thus, if bit 1 is clear in any of the EPT paging-structure entries used to translate the guest-physical address of a guest paging-structure entry, an attempt to use that entry to translate a linear address causes an EPT violation.

(This does not apply to loads of the PDPTe registers by the MOV to CR instruction for PAE paging; see Section 4.4.1. Those loads of guest PDPTes are treated as reads and do not cause EPT violations due to a guest-physical address not being writable.)

If the “sub-page write permissions for EPT” VM-execution control is 1, data writes to a guest-physical address that would cause an EPT violation (as indicated above) do not do so in certain situations. If the guest-physical address is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1, writes to certain 128-byte sub-pages may be allowed. See Section 28.2.4 for details.

- The access is an instruction fetch and the EPT paging structures prevent execute access to any of the bytes being fetched. Whether this occurs depends upon the setting of the “mode-based execute control for EPT” VM-execution control:
 - If the control is 0, an instruction fetch from a byte is prevented if bit 2 (execute access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.
 - If the control is 1, an instruction fetch from a byte is prevented in either of the following cases:
 - Paging maps the linear address of the byte as a supervisor-mode address and bit 2 (execute access for supervisor-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.

Paging maps a linear address as a supervisor-mode address if the U/S flag (bit 2) is 0 in at least one of the paging-structure entries controlling the translation of the linear address.
 - Paging maps the linear address of the byte as a user-mode address and bit 10 (execute access for user-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.

Paging maps a linear address as a user-mode address if the U/S flag is 1 in all of the paging-structure entries controlling the translation of the linear address. If paging is disabled (CR0.PG = 0), every linear address is a user-mode address.
- If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), the access is a supervisor shadow-stack access, and the EPT paging-structure entries used to translate the guest-physical address of the access disallow supervisor shadow-stack accesses. Such an access is disallowed if any of the following hold:
 - Bit 0 (read access) is clear in any EPT paging-structure entry used to translate the guest-physical address of the access.
 - Bit 1 (write access) is clear in any EPT paging-structure entry that references an EPT paging structure in the translation of the guest-physical address. (Clearing bit 1 in the EPT paging-structure entry that maps the page of the guest-physical address does not disallow shadow-stack reads and writes.)
 - Bit 60 (supervisor-shadow stack access) is clear in the EPT paging-structure entry that maps the page of the guest-physical address.

Supervisor shadow-stack control and the supervisor-shadow stack access bits in EPT paging-structure entries do not affect other accesses (including user shadow-stack accesses).

28.2.3.3 Prioritization of EPT Misconfigurations and EPT Violations

The translation of a linear address to a physical address requires one or more translations of guest-physical addresses using EPT (see Section 28.2.1). This section specifies the relative priority of EPT-induced VM exits with respect to each other and to other events that may be encountered when accessing memory using a linear address.

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

1. An EPT paging-structure entry is read (initially, this is an EPT PML4 entry):

1. This is a simplification of the more detailed description given in Section 28.2.2.

- a. If the entry is not present (see Section 28.2.2), an EPT violation occurs.
 - b. If the entry is present but its contents are not configured properly (see Section 28.2.3.1), an EPT misconfiguration occurs.
 - c. If the entry is present and its contents are configured properly, operation depends on whether the entry references another EPT paging structure (whether it is an EPT PDE with bit 7 set to 1 or an EPT PTE):
 - i) If the entry does reference another EPT paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
 - ii) Otherwise, the entry is used to produce the ultimate physical address (the translation of the original guest-physical address); step 2 is executed.
2. Once the ultimate physical address is determined, the privileges determined by the EPT paging-structure entries are evaluated:
- a. If the access to the guest-physical address is not allowed by these privileges (see Section 28.2.3.2), an EPT violation occurs.
 - b. If the access to the guest-physical address is allowed by these privileges, memory is accessed using the ultimate physical address.

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

1. An attempt is made to access a guest paging-structure entry with a guest-physical address (initially, the address in CR3 or PDPTTE register).
 - a. If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
 - b. If the access does not cause an EPT-induced VM exit, bit 0 (the present flag) of the entry is consulted:
 - i) If the present flag is 0 or any reserved bit is set, a page fault occurs.
 - ii) If the present flag is 1, no reserved bit is set, operation depends on whether the entry references another guest paging structure (whether it is a guest PDE with $PS = 1$ or a guest PTE):
 - If the entry does reference another guest paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
 - Otherwise, the entry is used to produce the ultimate guest-physical address (the translation of the original linear address); step 2 is executed.
2. Once the ultimate guest-physical address is determined, the privileges determined by the guest paging-structure entries are evaluated:
 - a. If the access to the linear address is not allowed by these privileges (e.g., it was a write to a read-only page), a page fault occurs.
 - b. If the access to the linear address is allowed by these privileges, an attempt is made to access memory at the ultimate guest-physical address:
 - i) If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
 - ii) If the access does not cause an EPT-induced VM exit, memory is accessed using the ultimate physical address (the translation, using EPT, of the ultimate guest-physical address).

If $CR0.PG = 0$, a linear address is treated as a guest-physical address and is translated using EPT (see above). This process, if it completes without an EPT violation or EPT misconfiguration, produces a physical address and determines the privileges allowed by the EPT paging-structure entries. If these privileges do not allow the access to the physical address (see Section 28.2.3.2), an EPT violation occurs. Otherwise, memory is accessed using the physical address.

28.2.4 Sub-Page Write Permissions

Section 28.2.3.2 explained how EPT enforces the access rights for guest-physical addresses using EPT violations. Since these access rights are determined using the EPT paging-structure entries that are used to translate a guest-physical address, their granularity is limited to that which is used to map pages (1-GByte, 2-MByte, or 4-KByte).

The **sub-page write-permission** feature allows the control of write accesses to guest-physical addresses to be controlled at finer granularity. Sub-page write permissions allow write accesses to be controlled at the granularity of naturally aligned 128-byte **sub-pages**. Specifically, the feature allows writes to selected sub-pages of 4-KByte page that would otherwise not be writable.

Sub-page write permissions are enabled setting the “sub-page write permissions for EPT” VM-execution control to 1. The remainder of this section describes changes to processor operation with this control setting.

Section 28.2.4.1 identifies the data accesses that are eligible for sub-page write permissions. Section 28.2.4.2 explains how the processor determines whether to allow such an access.

28.2.4.1 Write Accesses That Are Eligible for Sub-Page Write Permissions

A guest-physical address is eligible for sub-page write permissions if writes to it would be disallowed following Section 28.2.3.2: bit 1 (write access) is clear in any of the EPT paging-structure entries used to translate the guest-physical address. Guest-physical addresses to which writes would be disallowed for other reasons (e.g., the translation encounters an EPT paging-structure entry that is not present) are not eligible for sub-page write permissions.

In addition, a guest-physical address is eligible for sub-page write permissions only if it is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1. (Guest-physical addresses mapped with larger pages are not eligible for sub-page write permissions.)

For some memory accesses, the processor ignores bit 61 in an EPT PTE used to map a 4-KByte page and does not apply sub-page write permissions to the access. (In such a case, the access causes an EPT violation when indicated by the conditions given in Section 28.2.3.2.) Sub-page write permissions never apply to the following accesses:

- A write access performed within a transactional region.
- A write access by an enclave to an address within the enclave's ELRANGE. (Sub-page write permissions may apply to write accesses by an enclaves to addresses outside its ELRANGE.)
- A write access to the enclave page cache (EPC) by an Intel SGX instruction.
- A write access to a guest paging structure to update an accessed or dirty flag.
- Processor accesses to guest paging-structure entries when accessed and dirty flags for EPT are enabled (such accesses are treated as writes with regard to EPT violations).

There are additional accesses to which sub-page write permissions might not be applied (behavior is model-specific). The following items enumerate examples:

- A write access that crosses two 4-KByte pages. In this case, sub-page permissions may be applied to neither or to only one of the pages. (There is no write to either page unless the write is allowed to both pages.)
- A write access by an instruction that performs multiple write accesses (sub-page write permissions are intended principally for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR).

If a guest-physical address is eligible for sub-page write permissions, the processor determines whether to allow write to the address using the process described in Section 28.2.4.2.

If a guest-physical address is eligible for sub-page write permissions and that address translates to an address on the APIC-access page (see Section 29.4), the processor may treat a write access to the address as if the “virtualize APIC accesses” VM-execution control were 0. For that reason, it is recommended that software not configure any guest-physical address that translates to an address on the APIC-access page to be eligible for sub-page write permissions.

28.2.4.2 Determining an Access's Sub-Page Write Permission

Sub-page write permissions control write accesses individually to each of the 32 128-byte sub-pages of a 4-KByte page. Bits 11:7 of guest-physical address identify the sub-page.

For each guest-physical address eligible for sub-page write permissions, there is a 64-bit **sub-page permission vector (SPP vector)**. All addresses on a 4-KByte page use the same SPP vector. If an address's sub-page number (bits 11:7 of the address) is *S*, writes to address are allowed if and only if bit 2*S* of the sub-page permission is set to 1. (The bits at odd positions in a SPP vector are not used and must be zero.)

Each page's SPP vector is located in memory. For a write to a guest-physical address eligible for sub-page write permissions, the processor uses the following process to locate the address's SPP vector:

1. The SPPTP (sub-page-permission-table pointer) VM-execution control field contains the physical address of the 4-KByte root SPP table (SSPL4 table). Bits 47:39 of the guest-physical address identify a 64-bit entry in that table, called an SPPL4E.
2. A 4-KByte SPPL3 table is located at the physical address in the selected SPPL4E. Bits 38:30 of the guest-physical address identify a 64-bit entry in that table, called an SPPL3E.
3. A 4-KByte SPPL2 table is located at the physical address in the selected SPPL3E. Bits 29:21 of the guest-physical address identify a 64-bit entry in that table, called an SPPL2E.
4. A 4-KByte SPP-vector table (SSPL1 table) is located at the physical address in the selected SPPL2E. Bits 20:12 of the guest-physical address identify the 64-bit SPP vector for the address. As noted earlier, bit 2*S* of the sub-page permission vector determines whether the address may be written, where *S* is the value of address bits 11:7.

(The memory type used to access these tables is reported in bits 53:50 of the IA32_VMX_BASIC MSR. See Appendix A.1.)

A write access to multiple 128-byte sub-pages on a single 4-KByte page is allowed only if the indicated bit in the page's SPP vector for each of those sub-pages. The following items apply to cases in which an access writes to two 4-KByte pages:

- If a write to either page would be disallowed according to Section 28.2.3.2, the access might be disallowed even if the guest-physical address of that page is eligible for sub-page write permissions. (This behavior is model-specific.)
- The access is allowed only if, for each page, either (1) a write to the page would be allowed following Section 28.2.3.2; or (2) both (a) the guest-physical address of that page is eligible for sub-page write permissions; and (b) the page's sub-page vector allows the write (as described above).

Bit 0 of each entry (SPPL4E, SPPL3E, or SPPL2E) is the entry's **valid** bit. If the process above accesses an entry in which this bit is 0, the process stops and the logical processor incurs an **SPP miss**.

In each entry (SPPL4E, SPPL3E, or SPPL2E), bits 11:1 are reserved, as are bits 63:N, where N is the processor's physical-address width. If the process above accesses an entry in which the valid bit is 1 and in which some reserved bit is set, the process stops and the logical processor incurs an **SPP misconfiguration**. Bits in an SPP vector in odd positions are also reserved; an SPP misconfiguration occurs also any of those bits are set in the final SPP vector.

SPP misses and SPP misconfigurations are called **SPP-related events** and cause VM exits.

28.2.5 Accessed and Dirty Flags for EPT

The Intel 64 architecture supports **accessed and dirty flags** in ordinary paging-structure entries (see Section 4.8). Some processors also support corresponding flags in EPT paging-structure entries. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.

Software can enable accessed and dirty flags for EPT using bit 6 of the extended-page-table pointer (EPTP), a VM-execution control field (see Table 24-8 in Section 24.6.11). If this bit is 1, the processor will set the accessed and dirty flags for EPT as described below. In addition, setting this flag causes processor accesses to guest paging-structure entries to be treated as writes (see below and Section 28.2.3.2).

For any EPT paging-structure entry that is used during guest-physical-address translation, bit 8 is the accessed flag. For a EPT paging-structure entry that maps a page (as opposed to referencing another EPT paging structure), bit 9 is the dirty flag.

Whenever the processor uses an EPT paging-structure entry as part of guest-physical-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a guest-physical address, the processor sets the dirty flag (if it is not already set) in the EPT paging-structure entry that identifies the final physical address for the guest-physical address (either an EPT PTE or an EPT paging-structure entry in which bit 7 is 1).

When accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes (see Section 28.2.3.2). Thus, such an access will cause the processor to set the dirty flag in the EPT paging-structure entry that identifies the final physical address of the guest paging-structure entry.

(This does not apply to loads of the PDPT registers for PAE paging by the MOV to CR instruction; see Section 4.4.1. Those loads of guest PDPTs are treated as reads and do not cause the processor to set the dirty flag in any EPT paging-structure entry.)

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 EPT paging-structure entries in TLBs and paging-structure caches (see Section 28.3). 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 guest-physical address.

28.2.6 Page-Modification Logging

When accessed and dirty flags for EPT are enabled, software can track writes to guest-physical addresses using a feature called **page-modification logging**.

Software can enable page-modification logging by setting the “enable PML” VM-execution control (see Table 24-7 in Section 24.6.2). When this control is 1, the processor adds entries to the **page-modification log** as described below. The page-modification log is a 4-KByte region of memory located at the physical address in the PML address VM-execution control field. The page-modification log consists of 512 64-bit entries; the PML index VM-execution control field indicates the next entry to use.

Before allowing a guest-physical access, the processor may determine that it first needs to set an accessed or dirty flag for EPT (see Section 28.2.5). When this happens, the processor examines the PML index. If the PML index is not in the range 0–511, there is a **page-modification log-full event** and a VM exit occurs. In this case, the accessed or dirty flag is not set, and the guest-physical access that triggered the event does not occur.

If instead the PML index is in the range 0–511, the processor proceeds to update accessed or dirty flags for EPT as described in Section 28.2.5. If the processor updated a dirty flag for EPT (changing it from 0 to 1), it then operates as follows:

1. The guest-physical address of the access is written to the page-modification log. Specifically, the guest-physical address is written to physical address determined by adding 8 times the PML index to the PML address. Bits 11:0 of the value written are always 0 (the guest-physical address written is thus 4-KByte aligned).
2. The PML index is decremented by 1 (this may cause the value to transition from 0 to FFFFH).

Because the processor decrements the PML index with each log entry, the value may transition from 0 to FFFFH. At that point, no further logging will occur, as the processor will determine that the PML index is not in the range 0–511 and will generate a page-modification log-full event (see above).

28.2.7 EPT and Memory Typing

This section specifies how a logical processor determines the memory type use for a memory access while EPT is in use. (See Chapter 11, “Memory Cache Control” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details of memory typing in the Intel 64 architecture.) Section 28.2.7.1 explains how the memory type is determined for accesses to the EPT paging structures. Section 28.2.7.2 explains how the memory type is determined for an access using a guest-physical address that is translated using EPT.

28.2.7.1 Memory Type Used for Accessing EPT Paging Structures

This section explains how the memory type is determined for accesses to the EPT paging structures. The determination is based first on the value of bit 30 (cache disable—CD) in control register CR0:

- If CR0.CD = 0, the memory type used for any such reference is the EPT paging-structure memory type, which is specified in bits 2:0 of the extended-page-table pointer (EPTP), a VM-execution control field (see Section 24.6.11). A value of 0 indicates the uncacheable type (UC), while a value of 6 indicates the write-back type (WB). Other values are reserved.
- If CR0.CD = 1, the memory type used for any such reference is uncacheable (UC).

The MTRRs have no effect on the memory type used for an access to an EPT paging structure.

28.2.7.2 Memory Type Used for Translated Guest-Physical Addresses

The **effective memory type** of a memory access using a guest-physical address (an access that is translated using EPT) is the memory type that is used to access memory. The effective memory type is based on the value of bit 30 (cache disable—CD) in control register CR0; the **last** EPT paging-structure entry used to translate the guest-physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE); and the PAT memory type (see below):

- The **PAT memory type** depends on the value of CR0.PG:
 - If CR0.PG = 0, the PAT memory type is WB (writeback).¹
 - If CR0.PG = 1, the PAT memory type is the memory type selected from the IA32_PAT MSR as specified in Section 11.12.3, “Selecting a Memory Type from the PAT”.²
- The **EPT memory type** is specified in bits 5:3 of the last EPT paging-structure entry: 0 = UC; 1 = WC; 4 = WT; 5 = WP; and 6 = WB. Other values are reserved and cause EPT misconfigurations (see Section 28.2.3).
- If CR0.CD = 0, the effective memory type depends upon the value of bit 6 of the last EPT paging-structure entry:
 - If the value is 0, the effective memory type is the combination of the EPT memory type and the PAT memory type specified in Table 11-7 in Section 11.5.2.2, using the EPT memory type in place of the MTRR memory type.
 - If the value is 1, the memory type used for the access is the EPT memory type. The PAT memory type is ignored.
- If CR0.CD = 1, the effective memory type is UC.

The MTRRs have no effect on the memory type used for an access to a guest-physical address.

28.3 CACHING TRANSLATION INFORMATION

Processors supporting Intel® 64 and IA-32 architectures may accelerate the address-translation process by caching on the processor data from the structures in memory that control that process. Such caching is discussed in Section 4.10, “Caching Translation Information” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. The current section describes how this caching interacts with the VMX architecture.

The VPID and EPT features of the architecture for VMX operation augment this caching architecture. EPT defines the guest-physical address space and defines translations to that address space (from the linear-address space)

-
1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
 2. Table 11-11 in Section 11.12.3, “Selecting a Memory Type from the PAT” illustrates how the PAT memory type is selected based on the values of the PAT, PCD, and PWT bits in a page-table entry (or page-directory entry with PS = 1). For accesses to a guest paging-structure entry X, the PAT memory type is selected from the table by using a value of 0 for the PAT bit with the values of PCD and PWT from the paging-structure entry Y that references X (or from CR3 if X is in the root paging structure). With PAE paging, the PAT memory type for accesses to the PDPTs is WB.

and from that address space (to the physical-address space). Both features control the ways in which a logical processor may create and use information cached from the paging structures.

Section 28.3.1 describes the different kinds of information that may be cached. Section 28.3.2 specifies when such information may be cached and how it may be used. Section 28.3.3 details how software can invalidate cached information.

28.3.1 Information That May Be Cached

Section 4.10, “Caching Translation Information” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* identifies two kinds of translation-related information that may be cached by a logical processor: **translations**, which are mappings from linear page numbers to physical page frames, and **paging-structure caches**, which map the upper bits of a linear page number to information from the paging-structure entries used to translate linear addresses matching those upper bits.

The same kinds of information may be cached when VPIDs and EPT are in use. A logical processor may cache and use such information based on its function. Information with different functionality is identified as follows:

- **Linear mappings.**¹ There are two kinds:

- Linear translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
- Linear paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges. For example, bits 47:39 of a linear address would map to the address of the relevant page-directory-pointer table.

Linear mappings do not contain information from any EPT paging structure.

- **Guest-physical mappings.**² There are two kinds:

- Guest-physical translations. Each of these is a mapping from a guest-physical page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
- Guest-physical paging-structure-cache entries. Each of these is a mapping from the upper portion of a guest-physical address to the physical address of the EPT paging structure used to translate the corresponding region of the guest-physical address space, along with information about access privileges.

The information in guest-physical mappings about access privileges and memory typing is derived from EPT paging structures.

- **Combined mappings.**³ There are two kinds:

- Combined translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
- Combined paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges.

The information in combined mappings about access privileges and memory typing is derived from both guest paging structures and EPT paging structures.

Guest-physical mappings and combined mappings may also include SPP vectors and information about the data structures used to locate SPP vectors (see Section 28.2.4.2).

28.3.2 Creating and Using Cached Translation Information

The following items detail the creation of the mappings described in the previous section:⁴

-
1. Earlier versions of this manual used the term “VPID-tagged” to identify linear mappings.
 2. Earlier versions of this manual used the term “EPTP-tagged” to identify guest-physical mappings.
 3. Earlier versions of this manual used the term “dual-tagged” to identify combined mappings.

- The following items describe the creation of mappings while EPT is not in use (including execution outside VMX non-root operation):
 - Linear mappings may be created. They are derived from the paging structures referenced (directly or indirectly) by the current value of CR3 and are associated with the current VPID and the current PCID.
 - No linear mappings are created with information derived from paging-structure entries that are not present (bit 0 is 0) or that set reserved bits. For example, if a PTE is not present, no linear mapping are created for any linear page number whose translation would use that PTE.
 - No guest-physical or combined mappings are created while EPT is not in use.
- The following items describe the creation of mappings while EPT is in use:
 - Guest-physical mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by bits 51:12 of the current EPTP. These 40 bits contain the address of the EPT-PML4-table. (the notation **EP4TA** refers to those 40 bits). Newly created guest-physical mappings are associated with the current EP4TA.
 - Combined mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by the current EP4TA. If CR0.PG = 1, they are also derived from the paging structures referenced (directly or indirectly) by the current value of CR3. They are associated with the current VPID, the current PCID, and the current EP4TA.¹ No combined paging-structure-cache entries are created if CR0.PG = 0.²
 - No guest-physical mappings or combined mappings are created with information derived from EPT paging-structure entries that are not present (see Section 28.2.2) or that are misconfigured (see Section 28.2.3.1).
 - No combined mappings are created with information derived from guest paging-structure entries that are not present or that set reserved bits.
 - No linear mappings are created while EPT is in use.

The following items detail the use of the various mappings:

- If EPT is not in use (e.g., when outside VMX non-root operation), a logical processor may use cached mappings as follows:
 - For accesses using linear addresses, it may use linear mappings associated with the current VPID and the current PCID. It may also use global TLB entries (linear mappings) associated with the current VPID and any PCID.
 - No guest-physical or combined mappings are used while EPT is not in use.
- If EPT is in use, a logical processor may use cached mappings as follows:
 - For accesses using linear addresses, it may use combined mappings associated with the current VPID, the current PCID, and the current EP4TA. It may also use global TLB entries (combined mappings) associated with the current VPID, the current EP4TA, and any PCID.
 - For accesses using guest-physical addresses, it may use guest-physical mappings associated with the current EP4TA.
 - No linear mappings are used while EPT is in use.

4. This section associated cached information with the current VPID and PCID. If PCIDs are not supported or are not being used (e.g., because CR4.PCIDE = 0), all the information is implicitly associated with PCID 000H; see Section 4.10.1, “Process-Context Identifiers (PCIDs),” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

1. At any given time, a logical processor may be caching combined mappings for a VPID and a PCID that are associated with different EP4TAs. Similarly, it may be caching combined mappings for an EP4TA that are associated with different VPIDs and PCIDs.

2. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

28.3.3 Invalidating Cached Translation Information

Software modifications of paging structures (including EPT paging structures and the data structures used to locate SPP vectors) may result in inconsistencies between those structures and the mappings cached by a logical processor. Certain operations invalidate information cached by a logical processor and can be used to eliminate such inconsistencies.

28.3.3.1 Operations that Invalidate Cached Mappings

The following operations invalidate cached mappings as indicated:

- Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation (e.g., the INVLPG and INVPCID instructions) invalidate linear mappings and combined mappings.¹ They are required to do so only for the current VPID (but, for combined mappings, all EP4TAs). Linear mappings for the current VPID are invalidated even if EPT is in use.² Combined mappings for the current VPID are invalidated even if EPT is not in use.³
- An EPT violation invalidates any guest-physical mappings (associated with the current EP4TA) that would be used to translate the guest-physical address that caused the EPT violation. If that guest-physical address was the translation of a linear address, the EPT violation also invalidates any combined mappings for that linear address associated with the current PCID, the current VPID and the current EP4TA.
- If the “enable VPID” VM-execution control is 0, VM entries and VM exits invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs). Combined mappings for VPID 0000H are invalidated for all EP4TAs.
- Execution of the INVVPID instruction invalidates linear mappings and combined mappings. Invalidation is based on instruction operands, called the INVVPID type and the INVVPID descriptor. Four INVVPID types are currently defined:
 - **Individual-address.** If the INVVPID type is 0, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor and that would be used to translate the linear address specified in of the INVVPID descriptor. Linear mappings and combined mappings for that VPID and linear address are invalidated for all PCIDs and, for combined mappings, all EP4TAs. (The instruction may also invalidate mappings associated with other VPIDs and for other linear addresses.)
 - **Single-context.** If the INVVPID type is 1, the logical processor invalidates all linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EP4TAs. (The instruction may also invalidate mappings associated with other VPIDs.)
 - **All-context.** If the INVVPID type is 2, the logical processor invalidates linear mappings and combined mappings associated with all VPIDs except VPID 0000H and with all PCIDs. (The instruction may also invalidate linear mappings with VPID 0000H.) Combined mappings are invalidated for all EP4TAs.
 - **Single-context-retaining-globals.** If the INVVPID type is 3, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EP4TAs. The logical processor is **not** required to invalidate information that was used for **global** translations (although it may do so). See Section 4.10, “Caching Translation Information” for details regarding global translations. (The instruction may also invalidate mappings associated with other VPIDs.)

See Chapter 30 for details of the INVVPID instruction. See Section 28.3.3.3 for guidelines regarding use of this instruction.

-
1. See Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for an enumeration of operations that architecturally invalidate entries in the TLBs and paging-structure caches independent of VMX operation.
 2. While no linear mappings are created while EPT is in use, a logical processor may retain, while EPT is in use, linear mappings (for the same VPID as the current one) there were created earlier, when EPT was not in use.
 3. While no combined mappings are created while EPT is not in use, a logical processor may retain, while EPT is in not use, combined mappings (for the same VPID as the current one) there were created earlier, when EPT was in use.

- Execution of the INVEPT instruction invalidates guest-physical mappings and combined mappings. Invalidation is based on instruction operands, called the INVEPT type and the INVEPT descriptor. Two INVEPT types are currently defined:
 - **Single-context.** If the INVEPT type is 1, the logical processor invalidates all guest-physical mappings and combined mappings associated with the EP4TA specified in the INVEPT descriptor. Combined mappings for that EP4TA are invalidated for all VPIDs and all PCIDs. (The instruction may invalidate mappings associated with other EP4TAs.)
 - **All-context.** If the INVEPT type is 2, the logical processor invalidates guest-physical mappings and combined mappings associated with all EP4TAs (and, for combined mappings, for all VPIDs and PCIDs).
 See Chapter 30 for details of the INVEPT instruction. See Section 28.3.3.4 for guidelines regarding use of this instruction.
- A power-up or a reset invalidates all linear mappings, guest-physical mappings, and combined mappings.

28.3.3.2 Operations that Need Not Invalidate Cached Mappings

The following items detail cases of operations that are not required to invalidate certain cached mappings:

- Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation are not required to invalidate any guest-physical mappings.
- The INVVPID instruction is not required to invalidate any guest-physical mappings.
- The INVEPT instruction is not required to invalidate any linear mappings.
- VMX transitions are not required to invalidate any guest-physical mappings. If the “enable VPID” VM-execution control is 1, VMX transitions are not required to invalidate any linear mappings or combined mappings.
- The VMXOFF and VMXON instructions are not required to invalidate any linear mappings, guest-physical mappings, or combined mappings.

A logical processor may invalidate any cached mappings at any time. For this reason, the operations identified above may invalidate the indicated mappings despite the fact that doing so is not required.

28.3.3.3 Guidelines for Use of the INVVPID Instruction

The need for VMM software to use the INVVPID instruction depends on how that software is virtualizing memory (e.g., see Section 32.3, “Memory Virtualization”).

If EPT is not in use, it is likely that the VMM is virtualizing the guest paging structures. Such a VMM may configure the VMCS so that all or some of the operations that invalidate entries the TLBs and the paging-structure caches (e.g., the INVLPG instruction) cause VM exits. If VMM software is emulating these operations, it may be necessary to use the INVVPID instruction to ensure that the logical processor’s TLBs and the paging-structure caches are appropriately invalidated.

Requirements of when software should use the INVVPID instruction depend on the specific algorithm being used for page-table virtualization. The following items provide guidelines for software developers:

- Emulation of the INVLPG instruction may require execution of the INVVPID instruction as follows:
 - The INVVPID type is individual-address (0).
 - The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.
 - The linear address in the INVVPID descriptor is that of the operand of the INVLPG instruction being emulated.
- Some instructions invalidate all entries in the TLBs and paging-structure caches—except for global translations. An example is the MOV to CR3 instruction. (See Section 4.10, “Caching Translation Information” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details regarding global translations.) Emulation of such an instruction may require execution of the INVVPID instruction as follows:
 - The INVVPID type is single-context-retaining-globals (3).

- The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.
- Some instructions invalidate all entries in the TLBs and paging-structure caches—including for global translations. An example is the MOV to CR4 instruction if the value of value of bit 4 (page global enable—PGE) is changing. Emulation of such an instruction may require execution of the INVVPID instruction as follows:
 - The INVVPID type is single-context (1).
 - The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.

If EPT is not in use, the logical processor associates all mappings it creates with the current VPID, and it will use such mappings to translate linear addresses. For that reason, a VMM should not use the same VPID for different non-EPT guests that use different page tables. Doing so may result in one guest using translations that pertain to the other.

If EPT is in use, the instructions enumerated above might not be configured to cause VM exits and the VMM might not be emulating them. In that case, executions of the instructions by guest software properly invalidate the required entries in the TLBs and paging-structure caches (see Section 28.3.3.1); execution of the INVVPID instruction is not required.

If EPT is in use, the logical processor associates all mappings it creates with the value of bits 51:12 of current EPTP. If a VMM uses different EPTP values for different guests, it may use the same VPID for those guests. Doing so cannot result in one guest using translations that pertain to the other.

The following guidelines apply more generally and are appropriate even if EPT is in use:

- As detailed in Section 29.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the paging structures. If, at one time, the current VPID on a logical processor was a non-zero value X, it is recommended that software use the INVVPID instruction with the “single-context” INVVPID type and with VPID X in the INVVPID descriptor before a VM entry on the same logical processor that establishes VPID X and either (a) the “virtualize APIC accesses” VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.
- Software can use the INVVPID instruction with the “all-context” INVVPID type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from paging structures between separate uses of VMX operation.

28.3.3.4 Guidelines for Use of the INVEPT Instruction

The following items provide guidelines for use of the INVEPT instruction to invalidate information cached from the EPT paging structures.

- Software should use the INVEPT instruction with the “single-context” INVEPT type after making any of the following changes to an EPT paging-structure entry (the INVEPT descriptor should contain an EPTP value that references — directly or indirectly — the modified EPT paging structure):
 - Changing any of the privilege bits 2:0 from 1 to 0.¹
 - Changing the physical address in bits 51:12.
 - Clearing bit 8 (the accessed flag) if accessed and dirty flags for EPT will be enabled.
 - For an EPT PDPTTE or an EPT PDE, changing bit 7 (which determines whether the entry maps a page).
 - For the **last** EPT paging-structure entry used to translate a guest-physical address (an EPT PDPTTE with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), changing either bits 5:3 or bit 6. (These bits determine the effective memory type of accesses using that EPT paging-structure entry; see Section 28.2.7.)
 - For the **last** EPT paging-structure entry used to translate a guest-physical address (an EPT PDPTTE with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), clearing bit 9 (the dirty flag) if accessed and dirty flags for EPT will be enabled.

1. If the “mode-based execute control for EPT” VM-execution control is 1, software should use the INVEPT instruction after changing privilege bit 10 from 1 to 0.

- Software should use the INVEPT instruction with the “single-context” INVEPT type before a VM entry with an EPTP value X such that X[6] = 1 (accessed and dirty flags for EPT are enabled) if the logical processor had earlier been in VMX non-root operation with an EPTP value Y such that Y[6] = 0 (accessed and dirty flags for EPT are not enabled) and Y[51:12] = X[51:12].
- Software may use the INVEPT instruction after modifying a present EPT paging-structure entry (see Section 28.2.2) to change any of the privilege bits 2:0 from 0 to 1.¹ Failure to do so may cause an EPT violation that would not otherwise occur. Because an EPT violation invalidates any mappings that would be used by the access that caused the EPT violation (see Section 28.3.3.1), an EPT violation will not recur if the original access is performed again, even if the INVEPT instruction is not executed.
- Because a logical processor does not cache any information derived from EPT paging-structure entries that are not present (see Section 28.2.2) or misconfigured (see Section 28.2.3.1), it is not necessary to execute INVEPT following modification of an EPT paging-structure entry that had been not present or misconfigured.
- As detailed in Section 29.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the EPT paging structures. If EPT was in use on a logical processor at one time with EPTP X, it is recommended that software use the INVEPT instruction with the “single-context” INVEPT type and with EPTP X in the INVEPT descriptor before a VM entry on the same logical processor that enables EPT with EPTP X and either (a) the “virtualize APIC accesses” VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.
- Software can use the INVEPT instruction with the “all-context” INVEPT type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from EPT paging structures between separate uses of VMX operation.

In a system containing more than one logical processor, software must account for the fact that information from an EPT paging-structure entry may be cached on logical processors other than the one that modifies that entry. The process of propagating the changes to a paging-structure entry is commonly referred to as “TLB shutdown.” A discussion of TLB shutdown appears in Section 4.10.5, “Propagation of Paging-Structure Changes to Multiple Processors,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

1. If the “mode-based execute control for EPT” VM-execution control is 1, software may use the INVEPT instruction after modifying a present EPT paging-structure entry to change privilege bit 10 from 0 to 1.

31. Updates to Chapter 29, Volume 3C

Change bars show changes to Chapter 29 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

Changes to chapter: Minor updates regarding virtualized APIC registers as necessary.

CHAPTER 29

APIC VIRTUALIZATION AND VIRTUAL INTERRUPTS

The VMCS includes controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

When these controls are used, the processor will emulate many accesses to the APIC, track the state of the virtual APIC, and deliver virtual interrupts — all in VMX non-root operation with out a VM exit.¹

The processor tracks the state of the virtual APIC using a virtual-APIC page identified by the virtual-machine monitor (VMM). Section 29.1 discusses the virtual-APIC page and how the processor uses it to track the state of the virtual APIC.

The following are the VM-execution controls relevant to APIC virtualization and virtual interrupts (see Section 24.6 for information about the locations of these controls):

- **Virtual-interrupt delivery.** This control enables the evaluation and delivery of pending virtual interrupts (Section 29.2). It also enables the emulation of writes (memory-mapped or MSR-based, as enabled) to the APIC registers that control interrupt prioritization.
- **Use TPR shadow.** This control enables emulation of accesses to the APIC's task-priority register (TPR) via CR8 (Section 29.3) and, if enabled, via the memory-mapped or MSR-based interfaces.
- **Virtualize APIC accesses.** This control enables virtualization of memory-mapped accesses to the APIC (Section 29.4) by causing VM exits on accesses to a VMM-specified APIC-access page. Some of the other controls, if set, may cause some of these accesses to be emulated rather than causing VM exits.
- **Virtualize x2APIC mode.** This control enables virtualization of MSR-based accesses to the APIC (Section 29.5).
- **APIC-register virtualization.** This control allows memory-mapped and MSR-based reads of most APIC registers (as enabled) by satisfying them from the virtual-APIC page. It directs memory-mapped writes to the APIC-access page to the virtual-APIC page, following them by VM exits for VMM emulation.
- **Process posted interrupts.** This control allows software to post virtual interrupts in a data structure and send a notification to another logical processor; upon receipt of the notification, the target processor will process the posted interrupts by copying them into the virtual-APIC page (Section 29.6).

"Virtualize APIC accesses", "virtualize x2APIC mode", "virtual-interrupt delivery", and "APIC-register virtualization" are all secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, the processor operates as if these controls were all 0. See Section 24.6.2.

29.1 VIRTUAL APIC STATE

The **virtual-APIC page** is a 4-KByte region of memory that the processor uses to virtualize certain accesses to APIC registers and to manage virtual interrupts. The physical address of the virtual-APIC page is the **virtual-APIC address**, a 64-bit VM-execution control field in the VMCS (see Section 24.6.8).

Depending on the settings of certain VM-execution controls, the processor may virtualize certain fields on the virtual-APIC page with functionality analogous to that performed by the local APIC. Section 29.1.1 identifies and defines these fields. Section 29.1.2, Section 29.1.3, Section 29.1.4, and Section 29.1.5 detail the actions taken to virtualize updates to some of these fields.

With the exception of fields corresponding to virtualized APIC registers (defined in Section 29.1.1), software may modify the virtual-APIC page referenced by the current VMCS of a logical processor in VMX non-root operation. (This is an exception to the general requirement given in Section 24.11.4.)

1. In most cases, it is not necessary for a virtual-machine monitor (VMM) to inject virtual interrupts as part of VM entry.

29.1.1 Virtualized APIC Registers

Depending on the setting of certain VM-execution controls, a logical processor may virtualize certain accesses to APIC registers using the following fields on the virtual-APIC page:

- **Virtual task-priority register (VTPR):** the 32-bit field located at offset 080H on the virtual-APIC page.
- **Virtual processor-priority register (VPPR):** the 32-bit field located at offset 0A0H on the virtual-APIC page.
- **Virtual end-of-interrupt register (VEOI):** the 32-bit field located at offset 0B0H on the virtual-APIC page.
- **Virtual interrupt-service register (VISR):** the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H on the virtual-APIC page. Bit x of the VISR is at bit position $(x \& 1FH)$ at offset $(100H | ((x \& E0H) \gg 1))$. The processor uses only the low 4 bytes of each of the 16-byte fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H.
- **Virtual interrupt-request register (VIRR):** the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H on the virtual-APIC page. Bit x of the VIRR is at bit position $(x \& 1FH)$ at offset $(200H | ((x \& E0H) \gg 1))$. The processor uses only the low 4 bytes of each of the 16-Byte fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H.
- **Virtual interrupt-command register (VICR_LO):** the 32-bit field located at offset 300H on the virtual-APIC page
- **Virtual interrupt-command register (VICR_HI):** the 32-bit field located at offset 310H on the virtual-APIC page.

The VTPR field virtualizes the TPR whenever the “use TPR shadow” VM-execution control is 1. The other fields indicated above virtualize the corresponding APIC registers whenever the “virtual-interrupt delivery” VM-execution control is 1.

29.1.2 TPR Virtualization

The processor performs **TPR virtualization** in response to the following operations: (1) virtualization of the MOV to CR8 instruction; (2) virtualization of a write to offset 080H on the APIC-access page; and (3) virtualization of the WRMSR instruction with ECX = 808H. See Section 29.3, Section 29.4.3, and Section 29.5 for details of when TPR virtualization is performed.

The following pseudocode details the behavior of TPR virtualization:

```

IF “virtual-interrupt delivery” is 0
    THEN
        IF VTPR[7:4] < TPR threshold (see Section 24.6.8)
            THEN cause VM exit due to TPR below threshold;
        FI;
    ELSE
        perform PPR virtualization (see Section 29.1.3);
        evaluate pending virtual interrupts (see Section 29.2.1);
    FI;

```

Any VM exit caused by TPR virtualization is trap-like: the instruction causing TPR virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

29.1.3 PPR Virtualization

The processor performs **PPR virtualization** in response to the following operations: (1) VM entry; (2) TPR virtualization; and (3) EOI virtualization. See Section 26.3.2.5, Section 29.1.2, and Section 29.1.4 for details of when PPR virtualization is performed.

PPR virtualization uses the guest interrupt status (specifically, SVI; see Section 24.4.2) and VTPR. The following pseudocode details the behavior of PPR virtualization:

```

IF VTPR[7:4] ≥ SVI[7:4]

```

```

    THEN VPPR := VTPR & FFH;
    ELSE VPPR := SVI & FOH;
FI;

```

PPR virtualization always clears bytes 3:1 of VPPR.

PPR virtualization is caused only by TPR virtualization, EOI virtualization, and VM entry. Delivery of a virtual interrupt also modifies VPPR, but in a different way (see Section 29.2.2). No other operations modify VPPR, even if they modify SVI, VISR, or VTPR.

29.1.4 EOI Virtualization

The processor performs **EOI virtualization** in response to the following operations: (1) virtualization of a write to offset 0B0H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 80BH. See Section 29.4.3 and Section 29.5 for details of when EOI virtualization is performed. EOI virtualization occurs only if the “virtual-interrupt delivery” VM-execution control is 1.

EOI virtualization uses and updates the guest interrupt status (specifically, SVI; see Section 24.4.2). The following pseudocode details the behavior of EOI virtualization:

```

Vector := SVI;
VISR[Vector] := 0; (see Section 29.1.1 for definition of VISR)
IF any bits set in VISR
    THEN SVI := highest index of bit set in VISR
    ELSE SVI := 0;
FI;
perform PPR virtualization (see Section 29.1.3);
IF EOI_exit_bitmap[Vector] = 1 (see Section 24.6.8 for definition of EOI_exit_bitmap)
    THEN cause EOI-induced VM exit with Vector as exit qualification;
    ELSE evaluate pending virtual interrupts; (see Section 29.2.1)
FI;

```

Any VM exit caused by EOI virtualization is trap-like: the instruction causing EOI virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

29.1.5 Self-IPI Virtualization

The processor performs **self-IPI virtualization** in response to the following operations: (1) virtualization of a write to offset 300H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 83FH. See Section 29.4.3 and Section 29.5 for details of when self-IPI virtualization is performed. Self-IPI virtualization occurs only if the “virtual-interrupt delivery” VM-execution control is 1.

Each operation that leads to self-IPI virtualization provides an 8-bit vector (see Section 29.4.3 and Section 29.5). Self-IPI virtualization updates the guest interrupt status (specifically, RVI; see Section 24.4.2). The following pseudocode details the behavior of self-IPI virtualization:

```

VIRR[Vector] := 1; (see Section 29.1.1 for definition of VIRR)
RVI := max[RVI, Vector];
evaluate pending virtual interrupts; (see Section 29.2.1)

```

29.2 EVALUATION AND DELIVERY OF VIRTUAL INTERRUPTS

If the “virtual-interrupt delivery” VM-execution control is 1, certain actions in VMX non-root operation or during VM entry cause the processor to evaluate and deliver virtual interrupts.

Evaluation of virtual interrupts is triggered by certain actions change the state of the virtual-APIC page and is described in Section 29.2.1. This evaluation may result in recognition of a virtual interrupt. Once a virtual interrupt

is recognized, the processor may deliver it within VMX non-root operation without a VM exit. Virtual-interrupt delivery is described in Section 29.2.2.

29.2.1 Evaluation of Pending Virtual Interrupts

If the “virtual-interrupt delivery” VM-execution control is 1, certain actions cause a logical processor to **evaluate pending virtual interrupts**.

The following actions cause the evaluation of pending virtual interrupts: VM entry; TPR virtualization; EOI virtualization; self-IPI virtualization; and posted-interrupt processing. See Section 26.3.2.5, Section 29.1.2, Section 29.1.4, Section 29.1.5, and Section 29.6 for details of when evaluation of pending virtual interrupts is performed. No other operations cause the evaluation of pending virtual interrupts, even if they modify RVI or VPPR.

Evaluation of pending virtual interrupts uses the guest interrupt status (specifically, RVI; see Section 24.4.2). The following pseudocode details the evaluation of pending virtual interrupts:

```
IF “interrupt-window exiting” is 0 AND
RVI[7:4] > VPPR[7:4] (see Section 29.1.1 for definition of VPPR)
    THEN recognize a pending virtual interrupt;
ELSE
    do not recognize a pending virtual interrupt;
FI;
```

Once recognized, a virtual interrupt may be delivered in VMX non-root operation; see Section 29.2.2.

Evaluation of pending virtual interrupts is caused only by VM entry, TPR virtualization, EOI virtualization, self-IPI virtualization, and posted-interrupt processing. No other operations do so, even if they modify RVI or VPPR. The logical processor ceases recognition of a pending virtual interrupt following the delivery of a virtual interrupt.

29.2.2 Virtual-Interrupt Delivery

If a virtual interrupt has been recognized (see Section 29.2.1), it is delivered at an instruction boundary when the following conditions all hold: (1) RFLAGS.IF = 1; (2) there is no blocking by STI; (3) there is no blocking by MOV SS or by POP SS; and (4) the “interrupt-window exiting” VM-execution control is 0.

Virtual-interrupt delivery has the same priority as that of VM exits due to the 1-setting of the “interrupt-window exiting” VM-execution control.² Thus, non-maskable interrupts (NMIs) and higher priority events take priority over delivery of a virtual interrupt; delivery of a virtual interrupt takes priority over external interrupts and lower priority events.

Virtual-interrupt delivery wakes a logical processor from the same inactive activity states as would an external interrupt. Specifically, it wakes a logical processor from the states entered using the HLT and MWAIT instructions. It does not wake a logical processor in the shutdown state or in the wait-for-SIPI state.

Virtual-interrupt delivery updates the guest interrupt status (both RVI and SVI; see Section 24.4.2) and delivers an event within VMX non-root operation without a VM exit. The following pseudocode details the behavior of virtual-interrupt delivery (see Section 29.1.1 for definition of VISR, VIRR, and VPPR):

```
Vector := RVI;
VISR[Vector] := 1;
SVI := Vector;
VPPR := Vector & FOH;
VIRR[Vector] := 0;
IF any bits set in VIRR
    THEN RVI := highest index of bit set in VIRR
    ELSE RVI := 0;
FI;
deliver interrupt with Vector through IDT;
```

2. A logical processor never recognizes or delivers a virtual interrupt if the “interrupt-window exiting” VM-execution control is 1. Because of this, the relative priority of virtual-interrupt delivery and VM exits due to the 1-setting of that control is not defined.

cease recognition of any pending virtual interrupt;

If a logical processor is in enclave mode, an Asynchronous Enclave Exit (AEX) occurs before delivery of a virtual interrupt (see Chapter 39, “Enclave Exiting Events”).

29.3 VIRTUALIZING CR8-BASED TPR ACCESSES

In 64-bit mode, software can access the local APIC’s task-priority register (TPR) through CR8. Specifically, software uses the MOV from CR8 and MOV to CR8 instructions (see Section 10.8.6, “Task Priority in IA-32e Mode”). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these CR8-based APIC accesses by setting the “CR8-load exiting” and “CR8-store exiting” VM-execution controls, ensuring that the accesses cause VM exits (see Section 25.1.3). Alternatively, there are methods for virtualizing some CR8-based APIC accesses without VM exits.

Normally, an execution of MOV from CR8 or MOV to CR8 that does not fault or cause a VM exit accesses the APIC’s TPR. However, such an execution are treated specially if the “use TPR shadow” VM-execution control is 1. The following items provide details:

- **MOV from CR8.** The instruction loads bits 3:0 of its destination operand with bits 7:4 of VTPR (see Section 29.1.1). Bits 63:4 of the destination operand are cleared.
- **MOV to CR8.** The instruction stores bits 3:0 of its source operand into bits 7:4 of VTPR; the remainder of VTPR (bits 3:0 and bits 31:8) are cleared. Following this, the processor performs TPR virtualization (see Section 29.1.2).

29.4 VIRTUALIZING MEMORY-MAPPED APIC ACCESSES

When the local APIC is in xAPIC mode, software accesses the local APIC’s control registers using a memory-mapped interface. Specifically, software uses linear addresses that translate to physical addresses on page frame indicated by the base address in the IA32_APIC_BASE MSR (see Section 10.4.4, “Local APIC Status and Location”). This section describes how these accesses can be virtualized.

A virtual-machine monitor (VMM) can virtualize these memory-mapped APIC accesses by ensuring that any access to a linear address that would access the local APIC instead causes a VM exit. This could be done using paging or the extended page-table mechanism (EPT). Another way is by using the 1-setting of the “virtualize APIC accesses” VM-execution control.

If the “virtualize APIC accesses” VM-execution control is 1, the logical processor treats specially memory accesses using linear addresses that translate to physical addresses in the 4-KByte **APIC-access page**.^{3,4} (The APIC-access page is identified by the **APIC-access address**, a field in the VMCS; see Section 24.6.8.)

In general, an access to the APIC-access page causes an **APIC-access VM exit**. APIC-access VM exits provide a VMM with information about the access causing the VM exit. Section 29.4.1 discusses the priority of APIC-access VM exits.

Certain VM-execution controls enable the processor to virtualize certain accesses to the APIC-access page without a VM exit. In general, this virtualization causes these accesses to be made to the virtual-APIC page instead of the APIC-access page.

-
3. Even when addresses are translated using EPT (see Section 28.2), the determination of whether an APIC-access VM exit occurs depends on an access’s physical address, not its guest-physical address. Even when CRO.PG = 0, ordinary memory accesses by software use linear addresses; the fact that CRO.PG = 0 means only that the identity translation is used to convert linear addresses to physical (or guest-physical) addresses.
 4. If EPT is enabled and there is write to a guest-physical address that translates to an address on the APIC-access page that is eligible for sub-page write permissions (see Section 28.2.4.1), the processor may treat the write as if the “virtualize APIC accesses” VM-execution control were 0 (and not apply the treatment specified in this section). For that reason, it is recommended that software not configure any guest-physical address that translates to an address on the APIC-access page to be eligible for sub-page write permissions.

NOTES

Unless stated otherwise, this section characterizes only linear accesses to the APIC-access page; an access to the APIC-access page is a linear access if (1) it results from a memory access using a linear address; and (2) the access's physical address is the translation of that linear address. Section 29.4.6 discusses accesses to the APIC-access page that are not linear accesses.

The distinction between the APIC-access page and the virtual-APIC page allows a VMM to share paging structures or EPT paging structures among the virtual processors of a virtual machine (the shared paging structures referencing the same APIC-access address, which appears in the VMCS of all the virtual processors) while giving each virtual processor its own virtual APIC (the VMCS of each virtual processor will have a unique virtual-APIC address).

Section 29.4.2 discusses when and how the processor may virtualize read accesses from the APIC-access page. Section 29.4.3 does the same for write accesses. When virtualizing a write to the APIC-access page, the processor typically takes actions in addition to passing the write through to the virtual-APIC page.

The discussion in those sections uses the concept of an **operation** within which these memory accesses may occur. For those discussions, an "operation" can be an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.

The 1-setting of the "virtualize APIC accesses" VM-execution control may also affect accesses to the APIC-access page that do not result directly from linear addresses. This is discussed in Section 29.4.6.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 41.5.3 for details.

29.4.1 Priority of APIC-Access VM Exits

The following items specify the priority of APIC-access VM exits relative to other events.

- The priority of an APIC-access VM exit due to a memory access is below that of any page fault or EPT violation that that access may incur. That is, an access does not cause an APIC-access VM exit if it would cause a page fault or an EPT violation.
- A memory access does not cause an APIC-access VM exit until after the accessed flags are set in the paging structures (including EPT paging structures, if enabled).
- A write access does not cause an APIC-access VM exit until after the dirty flags are set in the appropriate paging structure and EPT paging structure (if enabled).
- With respect to all other events, any APIC-access VM exit due to a memory access has the same priority as any page fault or EPT violation that the access could cause. (This item applies to other events that the access may generate as well as events that may be generated by other accesses by the same operation.)

These principles imply, among other things, that an APIC-access VM exit may occur during the execution of a repeated string instruction (including INS and OUTS). Suppose, for example, that the first n iterations (n may be 0) of such an instruction do not access the APIC-access page and that the next iteration does access that page. As a result, the first n iterations may complete and be followed by an APIC-access VM exit. The instruction pointer saved in the VMCS references the repeated string instruction and the values of the general-purpose registers reflect the completion of n iterations.

29.4.2 Virtualizing Reads from the APIC-Access Page

A read access from the APIC-access page causes an APIC-access VM exit if any of the following are true:

- The "use TPR shadow" VM-execution control is 0.
- The access is for an instruction fetch.
- The access is more than 32 bits in size.
- The access is part of an operation for which the processor has already virtualized a write to the APIC-access page.

- The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access's address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a read access is virtualized depends on the setting of the "APIC-register virtualization" VM-execution control:

- If "APIC-register virtualization" and "virtual-interrupt delivery" VM-execution controls are both 0, a read access is virtualized if its page offset is 080H (task priority); otherwise, the access causes an APIC-access VM exit.
- If the "APIC-register virtualization" VM-execution control is 0 and the "virtual-interrupt delivery" VM-execution control is 1, a read access is virtualized if its page offset is 080H (task priority), 0B0H (end of interrupt), or 300H (interrupt command — low); otherwise, the access causes an APIC-access VM exit.
- If "APIC-register virtualization" is 1, a read access is virtualized if it is entirely within one the following ranges of offsets:
 - 020H–023H (local APIC ID);
 - 030H–033H (local APIC version);
 - 080H–083H (task priority);
 - 0B0H–0B3H (end of interrupt);
 - 0D0H–0D3H (logical destination);
 - 0E0H–0E3H (destination format);
 - 0F0H–0F3H (spurious-interrupt vector);
 - 100H–103H, 110H–113H, 120H–123H, 130H–133H, 140H–143H, 150H–153H, 160H–163H, or 170H–173H (in-service);
 - 180H–183H, 190H–193H, 1A0H–1A3H, 1B0H–1B3H, 1C0H–1C3H, 1D0H–1D3H, 1E0H–1E3H, or 1F0H–1F3H (trigger mode);
 - 200H–203H, 210H–213H, 220H–223H, 230H–233H, 240H–243H, 250H–253H, 260H–263H, or 270H–273H (interrupt request);
 - 280H–283H (error status);
 - 300H–303H or 310H–313H (interrupt command);
 - 320H–323H, 330H–333H, 340H–343H, 350H–353H, 360H–363H, or 370H–373H (LVT entries);
 - 380H–383H (initial count); or
 - 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

A read access from the APIC-access page that is virtualized returns data from the corresponding page offset on the virtual-APIC page.⁵

29.4.3 Virtualizing Writes to the APIC-Access Page

Whether a write access to the APIC-access page is virtualized depends on the settings of the VM-execution controls and the page offset of the access. Section 29.4.3.1 details when APIC-write virtualization occurs.

Unlike reads, writes to the local APIC have side effects; because of this, virtualization of writes to the APIC-access page may require emulation specific to the access's page offset (which identifies the APIC register being accessed). Section 29.4.3.2 describes this **APIC-write emulation**.

For some page offsets, it is necessary for software to complete the virtualization after a write completes. In these cases, the processor causes an **APIC-write VM exit** to invoke VMM software. Section 29.4.3.3 discusses APIC-write VM exits.

5. The memory type used for accesses that read from the virtual-APIC page is reported in bits 53:50 of the IA32_VMX_BASIC MSR (see Appendix A.1).

29.4.3.1 Determining Whether a Write Access is Virtualized

A write access to the APIC-access page causes an APIC-access VM exit if any of the following are true:

- The “use TPR shadow” VM-execution control is 0.
- The access is more than 32 bits in size.
- The access is part of an operation for which the processor has already virtualized a write (with a different page offset or a different size) to the APIC-access page.
- The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access’s address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a write access is virtualized depends on the settings of the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls:

- If the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls are both 0, a write access is virtualized if its page offset is 080H; otherwise, the access causes an APIC-access VM exit.
- If the “APIC-register virtualization” VM-execution control is 0 and the “virtual-interrupt delivery” VM-execution control is 1, a write access is virtualized if its page offset is 080H (task priority), 0B0H (end of interrupt), and 300H (interrupt command — low); otherwise, the access causes an APIC-access VM exit.
- If the “APIC-register virtualization” VM-execution control is 1, a write access is virtualized if it is entirely within one the following ranges of offsets:
 - 020H–023H (local APIC ID);
 - 080H–083H (task priority);
 - 0B0H–0B3H (end of interrupt);
 - 0D0H–0D3H (logical destination);
 - 0E0H–0E3H (destination format);
 - 0F0H–0F3H (spurious-interrupt vector);
 - 280H–283H (error status);
 - 300H–303H or 310H–313H (interrupt command);
 - 320H–323H, 330H–333H, 340H–343H, 350H–353H, 360H–363H, or 370H–373H (LVT entries);
 - 380H–383H (initial count); or
 - 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

The processor virtualizes a write access to the APIC-access page by writing data to the corresponding page offset on the virtual-APIC page.⁶ Following this, the processor performs certain actions after completion of the operation of which the access was a part.⁷ APIC-write emulation is described in Section 29.4.3.2.

29.4.3.2 APIC-Write Emulation

If the processor virtualizes a write access to the APIC-access page, it performs additional actions after completion of an operation of which the access was a part. These actions are called **APIC-write emulation**.

The details of APIC-write emulation depend upon the page offset of the virtualized write access:⁸

- 080H (task priority). The processor clears bytes 3:1 of VTPR and then causes TPR virtualization (Section 29.1.2).

6. The memory type used for accesses that write to the virtual-APIC page is reported in bits 53:50 of the IA32_VMX_BASIC MSR (see Appendix A.1).

7. Recall that, for the purposes of this discussion, an operation is an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.

8. For any operation, there can be only one page offset for which a write access was virtualized. This is because a write access is not virtualized if the processor has already virtualized a write access for the same operation with a different page offset.

- 0B0H (end of interrupt). If the “virtual-interrupt delivery” VM-execution control is 1, the processor clears VEOI and then causes EOI virtualization (Section 29.1.4); otherwise, the processor causes an APIC-write VM exit (Section 29.4.3.3).
- 300H (interrupt command — low). If the “virtual-interrupt delivery” VM-execution control is 1, the processor checks the value of VICR_LO to determine whether the following are all true:
 - Reserved bits (31:20, 17:16, 13) and bit 12 (delivery status) are all 0.
 - Bits 19:18 (destination shorthand) are 01B (self).
 - Bit 15 (trigger mode) is 0 (edge).
 - Bits 10:8 (delivery mode) are 000B (fixed).
 - Bits 7:4 (the upper half of the vector) are **not** 0000B.
 If all of the items above are true, the processor performs self-IPI virtualization using the 8-bit vector in byte 0 of VICR_LO (Section 29.1.5).
 If the “virtual-interrupt delivery” VM-execution control is 0, or if any of the items above are false, the processor causes an APIC-write VM exit (Section 29.4.3.3).
- 310H–313H (interrupt command — high). The processor clears bytes 2:0 of VICR_HI. No other virtualization or VM exit occurs.
- Any other page offset. The processor causes an APIC-write VM exit (Section 29.4.3.3).

APIC-write emulation takes priority over system-management interrupts (SMIs), INIT signals, and lower priority events. APIC-write emulation is not blocked if RFLAGS.IF = 0 or by the MOV SS, POP SS, or STI instructions.

If an operation causes a fault after a write access to the APIC-access page and before APIC-write emulation, and that fault is delivered without a VM exit, APIC-write emulation occurs after the fault is delivered and before the fault handler can execute. If an operation causes a VM exit (perhaps due to a fault) after a write access to the APIC-access page and before APIC-write emulation, the APIC-write emulation does not occur.

29.4.3.3 APIC-Write VM Exits

In certain cases, VMM software must be invoked to complete the virtualization of a write access to the APIC-access page. In this case, APIC-write emulation causes an **APIC-write VM exit**. (Section 29.4.3.2 details the cases that causes APIC-write VM exits.)

APIC-write VM exits are invoked by APIC-write emulation, and APIC-write emulation occurs after an operation that performs a write access to the APIC-access page. Because of this, every APIC-write VM exit is trap-like: it occurs after completion of the operation containing the write access that caused the VM exit (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

The basic exit reason for an APIC-write VM exit is “APIC write.” The exit qualification is the page offset of the write access that led to the VM exit.

As noted in Section 29.5, execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit if the “virtual-interrupt delivery” VM-execution control is 1. The exit qualification for such an APIC-write VM exit is 3F0H.

29.4.4 Instruction-Specific Considerations

Certain instructions that use linear address may cause page faults even though they do not use those addresses to access memory. The APIC-virtualization features may affect these instructions as well:

- **CLFLUSH, CLFLUSHOPT.** With regard to faulting, the processor operates as if each of these instructions reads from the linear address in its source operand. If that address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will flush the corresponding cache line on the virtual-APIC page instead of the APIC-access page.
- **ENTER.** With regard to faulting, the processor operates if ENTER writes to the byte referenced by the final value of the stack pointer (even though it does not if its size operand is non-zero). If that value translates to an

address on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will cause the APIC-write emulation appropriate to the address's page offset.

- **MASKMOVQ and MASKMOVDQU.** Even if the instruction's mask is zero, the processor may operate with regard to faulting as if MASKMOVQ or MASKMOVDQU writes to memory (the behavior is implementation-specific). In such a situation, an APIC-access VM exit may occur.
- **MONITOR.** With regard to faulting, the processor operates as if MONITOR reads from the effective address in RAX. If the resulting linear address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit.⁹ If it does not, it will monitor the corresponding address on the virtual-APIC page instead of the APIC-access page.
- **PREFETCH.** An execution of the PREFETCH instruction that would result in an access to the APIC-access page does not cause an APIC-access VM exit. Such an access may prefetch data; if so, it is from the corresponding address on the virtual-APIC page.

Virtualization of accesses to the APIC-access page is principally intended for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR. Use of an instruction that normally operates on floating-point, SSE, AVX, or AVX-512 registers may cause an APIC-access VM exit unconditionally regardless of the page offset it accesses on the APIC-access page.

29.4.5 Issues Pertaining to Page Size and TLB Management

The 1-setting of the "virtualize APIC accesses" VM-execution is guaranteed to apply only if translations to the APIC-access address use a 4-KByte page. The following items provide details:

- If EPT is not in use, any linear address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to the APIC-access page using a larger page may operate as if the "virtualize APIC accesses" VM-execution control were 0.
- If EPT is in use, any guest-physical address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to a guest-physical address that in turn translates to the APIC-access page using a larger page may operate as if the "virtualize APIC accesses" VM-execution control were 0. (This is true also for guest-physical accesses to the APIC-access page; see Section 29.4.6.1.)

In addition, software should perform appropriate TLB invalidation when making changes that may affect APIC-virtualization. The specifics depend on whether VPIDs or EPT is being used:

- **VPIDs being used but EPT not being used.** Suppose that there is a VPID that has been used before and that software has since made either of the following changes: (1) set the "virtualize APIC accesses" VM-execution control when it had previously been 0; or (2) changed the paging structures so that some linear address translates to the APIC-access address when it previously did not. In that case, software should execute INVVPID (see "INVVPID— Invalidate Translations Based on VPID" in Section 30.3) before performing on the same logical processor and with the same VPID.¹⁰
- **EPT being used.** Suppose that there is an EPTP value that has been used before and that software has since made either of the following changes: (1) set the "virtualize APIC accesses" VM-execution control when it had previously been 0; or (2) changed the EPT paging structures so that some guest-physical address translates to the APIC-access address when it previously did not. In that case, software should execute INVEPT (see "INVEPT— Invalidate Translations Derived from EPT" in Section 30.3) before performing on the same logical processor and with the same EPTP value.¹¹
- **Neither VPIDs nor EPT being used.** No invalidation is required.

9. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

10. INVVPID should use either (1) the all-contexts INVVPID type; (2) the single-context INVVPID type with the VPID in the INVVPID descriptor; or (3) the individual-address INVVPID type with the linear address and the VPID in the INVVPID descriptor.

11. INVEPT should use either (1) the global INVEPT type; or (2) the single-context INVEPT type with the EPTP value in the INVEPT descriptor.

Failure to perform the appropriate TLB invalidation may result in the logical processor operating as if the “virtualize APIC accesses” VM-execution control were 0 in responses to accesses to the affected address. (No invalidation is necessary if neither VPIDs nor EPT is being used.)

29.4.6 APIC Accesses Not Directly Resulting From Linear Addresses

Section 29.4 has described the treatment of accesses that use linear addresses that translate to addresses on the APIC-access page. This section considers memory accesses that do not result directly from linear addresses.

- An access is called a **guest-physical access** if (1) CR0.PG = 1;¹² (2) the “enable EPT” VM-execution control is 1;¹³ (3) the access’s physical address is the result of an EPT translation; and (4) either (a) the access was not generated by a linear address; or (b) the access’s guest-physical address is not the translation of the access’s linear address. Section 29.4.6.1 discusses the treatment of guest-physical accesses to the APIC-access page.
- An access is called a **physical access** if (1) either (a) the “enable EPT” VM-execution control is 0; or (b) the access’s physical address is not the result of a translation through the EPT paging structures; and (2) either (a) the access is not generated by a linear address; or (b) the access’s physical address is not the translation of its linear address. Section 29.4.6.2 discusses the treatment of physical accesses to the APIC-access page.

29.4.6.1 Guest-Physical Accesses to the APIC-Access Page

Guest-physical accesses include the following when guest-physical addresses are being translated using EPT:

- Reads from the guest paging structures when translating a linear address (such an access uses a guest-physical address that is not the translation of that linear address).
- Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).
- Updates to the accessed and dirty flags in the guest paging structures when using a linear address (such an access uses a guest-physical address that is not the translation of that linear address).
- Memory accesses by Intel Processor Trace when the “Intel PT uses guest physical addresses” VM-execution control is 1 (see Section 25.5.4).

Every guest-physical access using a guest-physical address that translates to an address on the APIC-access page causes an APIC-access VM exit. Such accesses are never virtualized regardless of the page offset.

The following items specify the priority relative to other events of APIC-access VM exits caused by guest-physical accesses to the APIC-access page.

- The priority of an APIC-access VM exit caused by a guest-physical access to memory is below that of any EPT violation that that access may incur. That is, a guest-physical access does not cause an APIC-access VM exit if it would cause an EPT violation.
- With respect to all other events, any APIC-access VM exit caused by a guest-physical access has the same priority as any EPT violation that the guest-physical access could cause.

29.4.6.2 Physical Accesses to the APIC-Access Page

Physical accesses include the following:

- If the “enable EPT” VM-execution control is 0:
 - Reads from the paging structures when translating a linear address.
 - Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).

12. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

13. “Enable EPT” 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 EPT” VM-execution control were 0. See Section 24.6.2.

- Updates to the accessed and dirty flags in the paging structures.
- If the “enable EPT” VM-execution control is 1, accesses to the EPT paging structures (including updates to the accessed and dirty flags for EPT).
- Any of the following accesses made by the processor to support VMX non-root operation:
 - Accesses to the VMCS region.
 - Accesses to data structures referenced (directly or indirectly) by physical addresses in VM-execution control fields in the VMCS. These include the I/O bitmaps, the MSR bitmaps, and the virtual-APIC page.
- Accesses that effect transitions into and out of SMM.¹⁴ These include the following:
 - Accesses to SMRAM during SMI delivery and during execution of RSM.
 - Accesses during SMM VM exits (including accesses to MSEG) and during VM entries that return from SMM.

A physical access to the APIC-access page may or may not cause an APIC-access VM exit. If it does not cause an APIC-access VM exit, it may access the APIC-access page or the virtual-APIC page. Physical write accesses to the APIC-access page may or may not cause APIC-write emulation or APIC-write VM exits.

The priority of an APIC-access VM exit caused by physical access is not defined relative to other events that the access may cause.

It is recommended that software not set the APIC-access address to any of the addresses used by physical memory accesses (identified above). For example, it should not set the APIC-access address to the physical address of any of the active paging structures if the “enable EPT” VM-execution control is 0.

29.5 VIRTUALIZING MSR-BASED APIC ACCESSES

When the local APIC is in x2APIC mode, software accesses the local APIC’s control registers using the MSR interface. Specifically, software uses the RDMSR and WRMSR instructions, setting ECX (identifying the MSR being accessed) to values in the range 800H–8FFH (see Section 10.12, “Extended XAPIC (x2APIC)”). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these MSR-based APIC accesses by configuring the MSR bitmaps (see Section 24.6.9) to ensure that the accesses cause VM exits (see Section 25.1.3). Alternatively, there are methods for virtualizing some MSR-based APIC accesses without VM exits.

Normally, an execution of RDMSR or WRMSR that does not fault or cause a VM exit accesses the MSR indicated in ECX. However, such an execution treats some values of ECX in the range 800H–8FFH specially if the “virtualize x2APIC mode” VM-execution control is 1. The following items provide details:

- **RDMSR.** The instruction’s behavior depends on the setting of the “APIC-register virtualization” VM-execution control.
 - If the “APIC-register virtualization” VM-execution control is 0, behavior depends upon the value of ECX.
 - If ECX contains 808H (indicating the TPR MSR), the instruction reads the 8 bytes from offset 080H on the virtual-APIC page (VTPR and the 4 bytes above it) into EDX:EAX. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not x2APIC mode).
 - If ECX contains any other value in the range 800H–8FFH, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a readable APIC register, EDX and EAX are loaded with the value of that register. If the local APIC is not in x2APIC mode or ECX does not indicate a readable APIC register, a general-protection fault occurs.
 - If “APIC-register virtualization” is 1 and ECX contains a value in the range 800H–8FFH, the instruction reads the 8 bytes from offset X on the virtual-APIC page into EDX:EAX, where $X = (ECX \& FFH) \ll 4$. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not in x2APIC mode).
- **WRMSR.** The instruction’s behavior depends on the value of ECX and the setting of the “virtual-interrupt delivery” VM-execution control.

14. Technically, these accesses do not occur in VMX non-root operation. They are included here for clarity.

Special processing applies in the following cases: (1) ECX contains 808H (indicating the TPR MSR); (2) ECX contains 80BH (indicating the EOI MSR) and the “virtual-interrupt delivery” VM-execution control is 1; and (3) ECX contains 83FH (indicating the self-IPI MSR) and the “virtual-interrupt delivery” VM-execution control is 1.

If special processing applies, no general-protection exception is produced due to the fact that the local APIC is in xAPIC mode. However, WRMSR does perform the normal reserved-bit checking:

- If ECX contains 808H or 83FH, a general-protection fault occurs if either EDX or EAX[31:8] is non-zero.
- If ECX contains 80BH, a general-protection fault occurs if either EDX or EAX is non-zero.

If there is no fault, WRMSR stores EDX:EAX at offset X on the virtual-APIC page, where $X = (ECX \& FFH) \ll 4$. Following this, the processor performs an operation depending on the value of ECX:

- If ECX contains 808H, the processor performs TPR virtualization (see Section 29.1.2).
- If ECX contains 80BH, the processor performs EOI virtualization (see Section 29.1.4).
- If ECX contains 83FH, the processor then checks the value of EAX[7:4] and proceeds as follows:
 - If the value is non-zero, the logical processor performs self-IPI virtualization with the 8-bit vector in EAX[7:0] (see Section 29.1.5).
 - If the value is zero, the logical processor causes an APIC-write VM exit as if there had been a write access to page offset 3F0H on the APIC-access page (see Section 29.4.3.3).

If special processing does not apply, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a writable APIC register, the value in EDX:EAX is written to that register. If the local APIC is not in x2APIC mode or ECX does not indicate a writable APIC register, a general-protection fault occurs.

29.6 POSTED-INTERRUPT PROCESSING

Posted-interrupt processing is a feature by which a processor processes the virtual interrupts by recording them as pending on the virtual-APIC page.

Posted-interrupt processing is enabled by setting the “process posted interrupts” VM-execution control. The processing is performed in response to the arrival of an interrupt with the **posted-interrupt notification vector**. In response to such an interrupt, the processor processes virtual interrupts recorded in a data structure called a **posted-interrupt descriptor**. The posted-interrupt notification vector and the address of the posted-interrupt descriptor are fields in the VMCS; see Section 24.6.8.

If the “process posted interrupts” VM-execution control is 1, a logical processor uses a 64-byte posted-interrupt descriptor located at the posted-interrupt descriptor address. The posted-interrupt descriptor has the following format:

Table 29-1. Format of Posted-Interrupt Descriptor

Bit Position(s)	Name	Description
255:0	Posted-interrupt requests	One bit for each interrupt vector. There is a posted-interrupt request for a vector if the corresponding bit is 1
256	Outstanding notification	If this bit is set, there is a notification outstanding for one or more posted interrupts in bits 255:0
511:257	Reserved for software and other agents	These bits may be used by software and by other agents in the system (e.g., chipset). The processor does not modify these bits.

The notation **PIR** (posted-interrupt requests) refers to the 256 posted-interrupt bits in the posted-interrupt descriptor.

Use of the posted-interrupt descriptor differs from that of other data structures that are referenced by pointers in a VMCS. There is a general requirement that software ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. That requirement does

not apply to the posted-interrupt descriptor. There is a requirement, however, that such modifications be done using locked read-modify-write instructions.

If the “external-interrupt exiting” VM-execution control is 1, any unmasked external interrupt causes a VM exit (see Section 25.2). If the “process posted interrupts” VM-execution control is also 1, this behavior is changed and the processor handles an external interrupt as follows:¹⁵

1. The local APIC is acknowledged; this provides the processor core with an interrupt vector, called here the **physical vector**.
2. If the physical vector equals the posted-interrupt notification vector, the logical processor continues to the next step. Otherwise, a VM exit occurs as it would normally due to an external interrupt; the vector is saved in the VM-exit interruption-information field.
3. The processor clears the outstanding-notification bit in the posted-interrupt descriptor. This is done atomically so as to leave the remainder of the descriptor unmodified (e.g., with a locked AND operation).
4. The processor writes zero to the EOI register in the local APIC; this dismisses the interrupt with the posted-interrupt notification vector from the local APIC.
5. The logical processor performs a logical-OR of PIR into VIRR and clears PIR. No other agent can read or write a PIR bit (or group of bits) between the time it is read (to determine what to OR into VIRR) and when it is cleared.
6. The logical processor sets RVI to be the maximum of the old value of RVI and the highest index of all bits that were set in PIR; if no bit was set in PIR, RVI is left unmodified.
7. The logical processor evaluates pending virtual interrupts as described in Section 29.2.1.

The logical processor performs the steps above in an uninterruptible manner. If step #7 leads to recognition of a virtual interrupt, the processor may deliver that interrupt immediately.

Steps #1 to #7 above occur when the interrupt controller delivers an unmasked external interrupt to the CPU core. The following items consider certain cases of interrupt delivery:

- Interrupt delivery can occur between iterations of a REP-prefixed instruction (after at least one iteration has completed but before all iterations have completed). If this occurs, the following items characterize processor state after posted-interrupt processing completes and before guest execution resumes:
 - RIP references the REP-prefixed instruction;
 - RCX, RSI, and RDI are updated to reflect the iterations completed; and
 - RFLAGS.RF = 1.
- Interrupt delivery can occur when the logical processor is in the active, HLT, or MWAIT states. If the logical processor had been in the active or MWAIT state before the arrival of the interrupt, it is in the active state following completion of step #7; if it had been in the HLT state, it returns to the HLT state after step #7 (if a pending virtual interrupt was recognized, the logical processor may immediately wake from the HLT state).
- Interrupt delivery can occur while the logical processor is in enclave mode. If the logical processor had been in enclave mode before the arrival of the interrupt, an Asynchronous Enclave Exit (AEX) may occur before the steps #1 to #7 (see Chapter 39, “Enclave Exiting Events”). If no AEX occurs before step #1 and a VM exit occurs at step #2, an AEX occurs before the VM exit is delivered.

15. VM entry ensures that the “process posted interrupts” VM-execution control is 1 only if the “external-interrupt exiting” VM-execution control is also 1. See Section 26.2.1.1.

32. 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*.

Changes to this chapter: Minor updates and corrections as necessary.

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.

Intel Processor Trace was first introduced in Intel® processors based on Broadwell microarchitecture and Intel Atom® processors based on Goldmont microarchitecture. 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. Further, Precise Event-Based Sampling (PEBS) can be configured to log PEBS records in the Intel PT trace; see Section 18.5.5.2.

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.
- Packets containing groups of processor state values:
 - Block Begin Packets (BBP): Indicate the type of state held in the following group.
 - Block Item Packets (BIP): Indicate the state values held in the group.
 - Block End Packets (BEP): Indicate the end of the current group.

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
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	INT1, INT3, INT <i>n</i> , 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 sub-categories:

- **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 35-24 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 to 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 35-41 “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. Support for PTWRITE is introduced in Intel® Atom™ processors based on the Goldmont Plus microarchitecture.

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]. Support for Power Event Tracing is introduced in Intel® Atom™ processors based on the Goldmont Plus microarchitecture.

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
<pre> 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 </pre>	<pre> TIP.PGE(RangeBase) TNT(0) TIP.PGD(RangeLimit+1) </pre>

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. 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 book-keeping 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_ENCLAVE1) 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.

1. Trace packets generation is disabled in a production enclave, see Section 35.2.8.5. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

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 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-cacheable (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[31:0] > 0.
- Illegal Output Offset
IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than the mask value IA32_RTIT_OUTPUT_MASK_PTRS[31:0].

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 the value from bits 31:7 (MaskOrTableOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS into bits 27:3 of proc_trace_table_offset. 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.

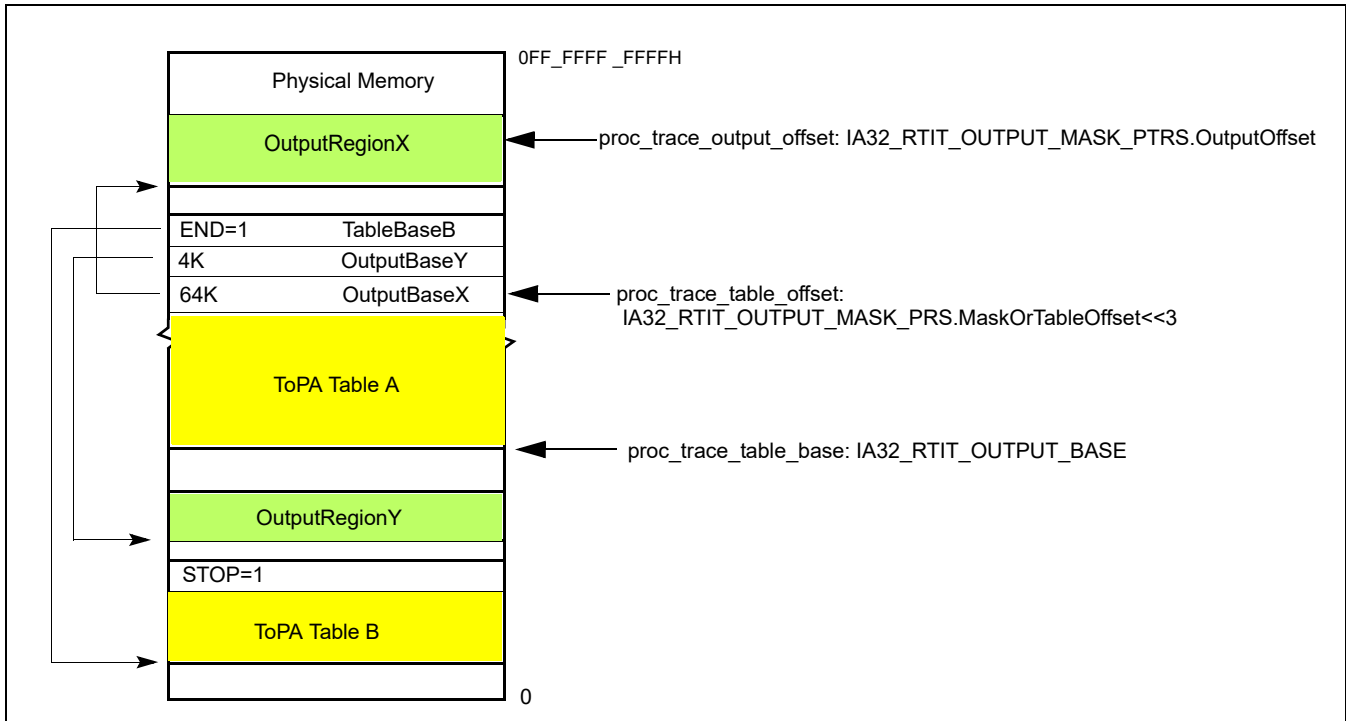


Figure 35-1. ToPA Memory Illustration

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:

$$\text{trace_store_phys_addr} := \text{Base address from current ToPA table entry} + \text{proc_trace_output_offset}$$

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 Table 35-3 and the section that follows 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 = 0FFFFFF8H`). 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 the output MSR values account for all packets generated to that point, after which the processor will cease updating the output MSR values 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].

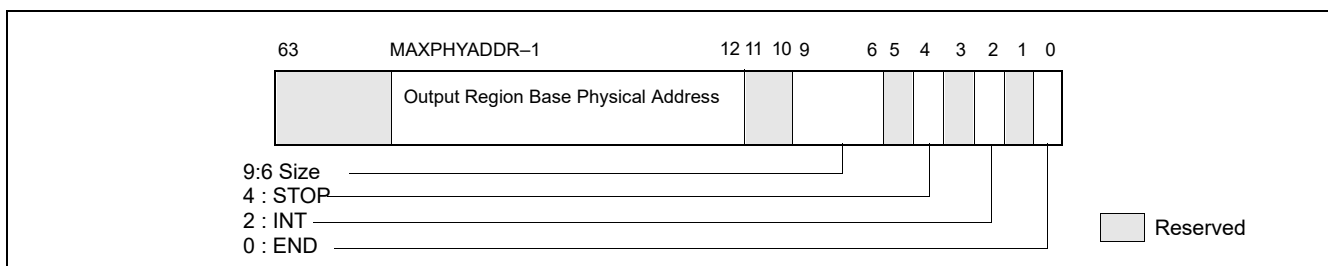


Figure 35-2. Layout of ToPA Table Entry

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.

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
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.
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.

PMI Preservation

In some cases a ToPA PMI may be taken after completion of an XSAVES instruction that saves 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, the PMI Preservation feature has been added. Support for this feature is indicated by CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6].

When IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, PMI preservation is enabled. When a ToPA region with INT=1 is filled, a PMI is pended and the new IA32_RTIT_STATUS.PendToPAPMI[7] is set to 1. If this bit is set when Intel PT is enabled, such that IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a ToPA PMI is pended. This behavior ensures that any ToPA PMI that is pended during XSAVES, and hence can't be properly handled, will be re-pended when the saved PT state is restored.

When this feature is enabled, the PMI handler should take the following actions:

1. Ignore ToPA PMIs that are taken when TraceEn = 0. This indicates that the PMI was pended during Intel PT disable, and the PendToPAPMI flag will ensure that the PMI is re-pended once Intel PT is re-enabled in the same context. For this reason, the PendToPAPMI bit should be left set to 1.
2. If TraceEn=1 and the PMI can be properly handled, clear the new PendTopaPMI bit. This will ensure that additional, spurious ToPA PMIs are not taken. It is required that PendToPAPMI is cleared before the PMI LVT mask is cleared in the APIC, and before any clearing of either LBRs_FROZEN or COUNTERS_FROZEN in IA32_PERF_GLOBAL_STATUS.

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 MSR 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. On processors that support PMI preservation (CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1), setting IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PMI that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendTopaPMI[7] = 1. A PMI will then be pended when the saved PT context is later restored.

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, if PMI preservation is not in use. 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
<pre> 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" 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; </pre>

ToPA Errors

When a malformed ToPA entry is found, an **operational 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), for ToPA entries with END=0.
 - c. ToPA entry base address sets upper physical address bits not supported by the processor.

3. **Illegal ToPA Output Offset.**

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 35.4.2.26.

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 write pointer (IA32_RTIT_OUTPUT_BASE + proc_trace_table_offset) 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 warm or 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. 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. This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 4] ("PTWRITE Supported") is 0.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
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. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 0] (“CR3 Filtering Support”) is 0.
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 35-41 “PTW Packet Definition”). This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 4] (“PTWRITE Supported”) is 0.
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 35.2.5.4 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 $2^{(\text{CycThresh}-1)}$. 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_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.RANGE CNT[2:0] < 1.
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.RANGE CNT[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.RANGE CNT[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.
55:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
56	InjectPsbPmiOnEnable	0	1: Enables use of IA32_RTIT_STATUS bits PendPSB[6] and PendTopaPMI[7], see Section 35.2.7.4, "IA32_RTIT_STATUS MSR" for behavior of these bits. 0: IA32_RTIT_STATUS bits 6 and 7 are ignored. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 0.
59:57	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 35.4.1.1).

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, though some fields cannot be modified by software. See Table 35-7 for details. 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.
6	PendPSB	0	If IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, the processor sets this bit when the threshold for a PSB+ to be inserted has been reached. The processor will clear this bit when the PSB+ has been inserted into the trace. If PendPSB = 1 and InjectPsbPmiOnEnable = 1 when IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a PSB+ will be inserted into the trace. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1.
7	PendTopaPMI	0	If IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, the processor sets this bit when the threshold for a ToPA PMI to be inserted has been reached. Software should clear this bit once the ToPA PMI has been handled, see “ToPA PMI” for details. If PendTopaPMI = 1 and InjectPsbPmiOnEnable = 1 when IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a PMI will be pended. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1.
31:8	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. This field is reserved when CPUID.(EAX=14H,ECX=0):EBX[bit 1] (“Configurable PSB and CycleAccurate Mode Supported”) is 0.
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 in canonical form, otherwise a **general-protection fault (#GP)** fault will result.

Each MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a **general-protection fault (#GP)**.

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 is accessible if CPUID.(EAX=14H, ECX=0):EBX[bit 0], “CR3 Filtering Support”, is 1. 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	<p>The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>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.</p> <p>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).</p> <p>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.</p>
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	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>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.</p> <p>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.</p>

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>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 MaskOffsetTableOffset field, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.</p> <p>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.</p>

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: <ul style="list-style-type: none"> ▪ 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.

35.2.8.2 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.

35.2.8.3 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.

35.2.8.4 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 write IA32_RTIT_CTL in VMX operation 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 35.4.2.26.

35.2.8.5 Intel® Software Guard Extensions (Intel® SGX)

Intel SGX provides an application with the ability to instantiate a protective container (an enclave) with confidentiality and integrity (see the *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and Intel SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. An 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

Intel SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see the *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by an enclave entry or exit. Hence, the generation of ContextEn-dependent packets within a debug enclave is allowed.

35.2.8.6 SENTER/ENTERACCS and ACM

GETSEC[SENDER] 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.

35.2.8.7 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 MaskOffsetTableOffset 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 bits 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_CTL.MTCFreq with unsupported encoding will cause #GP fault.
EBX	15:0	Bitmap of supported Cycle Threshold values	The non-zero bits indicate the map of supported encoding values 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 bits indicate the map of supported encoding values 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_LIP, MSR_LER_TO_LIP
- 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 Information 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 from 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 from 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+.

A PSB+ can be lost in some scenarios. If IA32_RTIT_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32_RTIT_CTL.TraceEn, a VM-exit that clears IA32_RTIT_CTL.TraceEn, an #SMI, or any time that either IA32_RTIT_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32_RTIT_STATUS.Error is set (e.g., by an Intel PT output error). On processors that support PSB preservation (CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1), setting IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PSB+ that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendPSB[6] = 1. A PSB will be inserted, and PendPSB cleared, when PT is later re-enabled while PendPSB = 1.

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+. If IA32_RTIT_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32_RTIT_CTL.TraceEn, a VM-exit that clears IA32_RTIT_CTL.TraceEn, an #SMI, or any time that either IA32_RTIT_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32_RTIT_STATUS.Error is set (e.g., by an Intel PT output error). On processors that support PSB preservation (CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1), setting IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PSB+ that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendPSB[6] = 1. A PSB will then be pended when the saved PT context is later restored.

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. If the overflow resolves while PacketEn=1, only timing packets may come between the OVF and the FUP. If the overflow resolves while PacketEn=0, any other packets that are not dependent on PacketEn may come between the OVF and 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 35.4.1.1 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 35.4.1.1 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.

35.4.1.1 Packet Blocks

Packet blocks are a means to dump one or more groups of state values. Packet blocks begin with a Block Begin Packet (BBP), which indicates what type of state is held within the block. Following each BBP there may be one or more Block Item Packets (BIPs), which contain the state values. The block is terminated by either a Block End Packet (BEP) or another BBP indicating the start of a new block.

The BIP packet includes an ID value that, when combined with the Type field from the BBP that preceded it, uniquely identifies the state value held in the BIP payload. The size of each BIP packet payload is provided by the Size field in the preceding BBP packet.

Each block type can have up to 32 items defined for it. There is no guarantee, however, that each block of that type will hold all 32 items. For more details on which items to expect, see documentation on the specific block type of interest.

See the BBP packet description (Section 35.4.2.26) for details on packet block generation scenarios.

Packet blocks are entirely generated within an instruction or between instructions, which dictates the types of packets (aside from BIPs) that may be seen within a packet block. Packets that indicate control flow changes, or other indication of instruction completion, cannot be generated within a block. These are listed in the following table. Other packets, including timing packets, may occur between BBP and BEP.

Table 35-16. Packets Forbidden Between BBP and BEP

TNT
TIP, TIP.PGE, TIP.PGD
MODE.Exec, MODE.TSX
PIP, VMCS
TraceStop
PSB, PSBEND
PTW
MWAIT

It is possible to encounter an internal buffer overflow in the middle of a block. In such a case, it is guaranteed that packet generation will not resume in the middle of a block, and hence the OVF packet terminates the current block. Depending on the duration of the overflow, subsequent blocks may also be lost.

Decoder Implications

When a Block Begin Packet (BBP) is encountered, the decoder will need to decode some packets within the block differently from those outside a block. The Block Item Packet (BIP) header byte has the same encoding as a TNT packet outside of a block, but must be treated as a BIP header (with following payload) within one.

When an OVF packet is encountered, the decoder should treat that as a block ending condition. Packet generation will not resume within a block.

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.

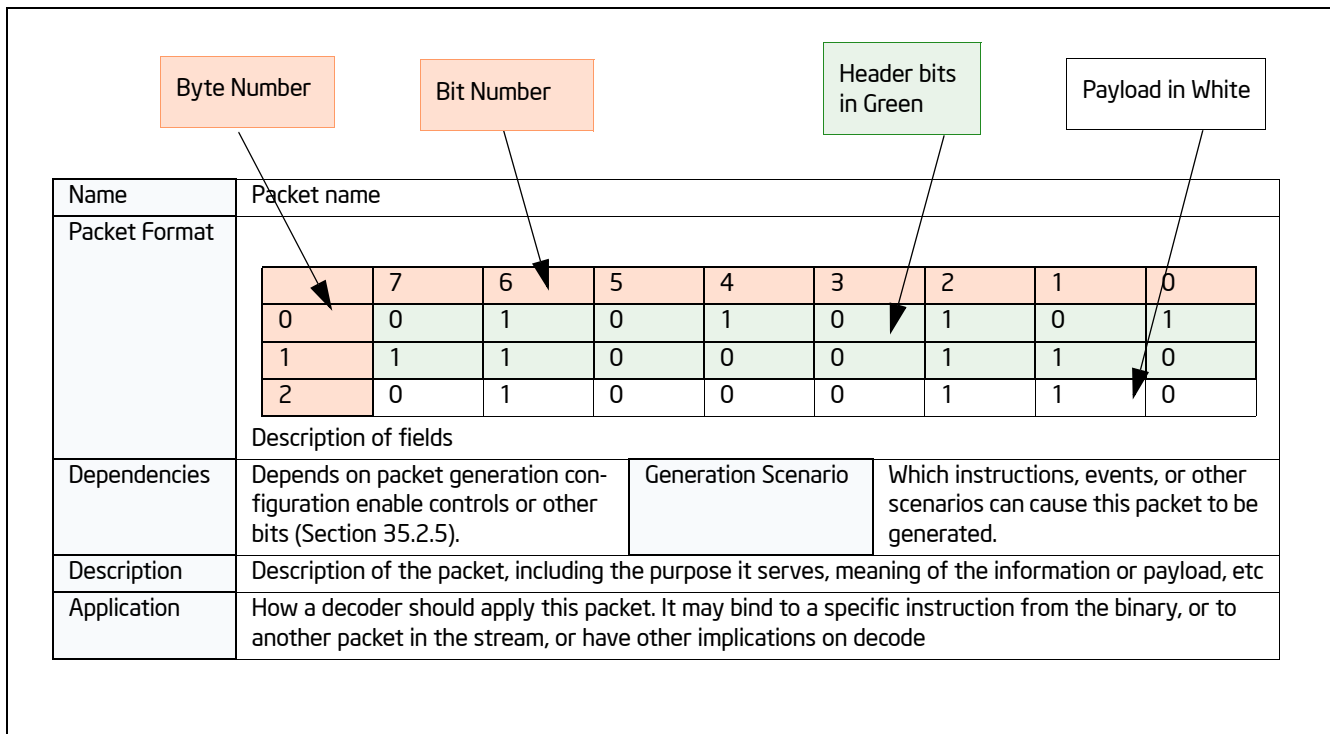


Figure 35-3. Interpreting Tabular Definition of Packet Format

35.4.2.1 Taken/Not-taken (TNT) Packet

Table 35-17. TNT Packet Definition

Name	Taken/Not-taken (TNT) Packet																																																																																											
Packet Format	<table border="1" style="width:100%; text-align:center;"> <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><td></td> </tr> <tr> <td>0</td><td>1</td><td>B₁</td><td>B₂</td><td>B₃</td><td>B₄</td><td>B₅</td><td>B₆</td><td>0</td><td>Short TNT</td> </tr> </table>										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																																																															
		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																																																																																			
	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 from 1 to 47 TNT bits.																																																																																											
	<table border="1" style="width:100%; text-align:center;"> <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><td></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><td rowspan="8">Long TNT</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>1</td> </tr> <tr> <td>2</td><td>B₄₀</td><td>B₄₁</td><td>B₄₂</td><td>B₄₃</td><td>B₄₄</td><td>B₄₅</td><td>B₄₆</td><td>B₄₇</td> </tr> <tr> <td>3</td><td>B₃₂</td><td>B₃₃</td><td>B₃₄</td><td>B₃₅</td><td>B₃₆</td><td>B₃₇</td><td>B₃₈</td><td>B₃₉</td> </tr> <tr> <td>4</td><td>B₂₄</td><td>B₂₅</td><td>B₂₆</td><td>B₂₇</td><td>B₂₈</td><td>B₂₉</td><td>B₃₀</td><td>B₃₁</td> </tr> <tr> <td>5</td><td>B₁₆</td><td>B₁₇</td><td>B₁₈</td><td>B₁₉</td><td>B₂₀</td><td>B₂₁</td><td>B₂₂</td><td>B₂₃</td> </tr> <tr> <td>6</td><td>B₈</td><td>B₉</td><td>B₁₀</td><td>B₁₁</td><td>B₁₂</td><td>B₁₃</td><td>B₁₄</td><td>B₁₅</td> </tr> <tr> <td>7</td><td>1</td><td>B₁</td><td>B₂</td><td>B₃</td><td>B₄</td><td>B₅</td><td>B₆</td><td>B₇</td> </tr> </table>										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
	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																																																																																				
	<p>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.</p> <table border="1" style="width:100%; text-align:center;"> <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><td></td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>B₁</td><td>B₂</td><td>B₃</td><td>B₄</td><td>0</td><td>Short TNT</td> </tr> </table>										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	1	B ₁	B ₂	B ₃	B ₄	0	Short TNT																																																																																			
	<table border="1" style="width:100%; text-align:center;"> <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><td></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><td rowspan="8">Long TNT</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>1</td> </tr> <tr> <td>2</td><td>B₂₄</td><td>B₂₅</td><td>B₂₆</td><td>B₂₇</td><td>B₂₈</td><td>B₂₉</td><td>B₃₀</td><td>B₃₁</td> </tr> <tr> <td>3</td><td>B₁₆</td><td>B₁₇</td><td>B₁₈</td><td>B₁₉</td><td>B₂₀</td><td>B₂₁</td><td>B₂₂</td><td>B₂₃</td> </tr> <tr> <td>4</td><td>B₈</td><td>B₉</td><td>B₁₀</td><td>B₁₁</td><td>B₁₂</td><td>B₁₃</td><td>B₁₄</td><td>B₁₅</td> </tr> <tr> <td>5</td><td>1</td><td>B₁</td><td>B₂</td><td>B₃</td><td>B₄</td><td>B₅</td><td>B₆</td><td>B₇</td> </tr> <tr> <td>6</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> <tr> <td>7</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> </table>										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
	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.																																																																																							

Table 35-17. TNT Packet Definition (Contd.)

Description	<p>Provides the taken/not-taken results for the last 1..6 (Short TNT) or 1..47 (Long TNT) conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 35.4.2.2). The TNT payload bits should be interpreted as follows:</p> <ul style="list-style-type: none"> ▪ 1 indicates a taken conditional branch, or a compressed RET ▪ 0 indicates a not-taken conditional branch <p>TNT payload bits are stored internal to the processor in a TNT buffer, until either the buffer is filled or another packet is to be generated. In either case a TNT packet holding the buffered bits will be emitted, and the TNT buffer will be marked as empty.</p>
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 35-18. IP Packet Definition

Name	Target IP (TIP) Packet																																																																																												
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table>				7	6	5	4	3	2	1	0	0	IPBytes			0	1	1	0	1	1	TargetIP[7:0]								2	TargetIP[15:8]								3	TargetIP[23:16]								4	TargetIP[31:24]								5	TargetIP[39:32]								6	TargetIP[47:40]								7	TargetIP[55:48]								8	TargetIP[63:56]							
	7	6	5	4	3	2	1	0																																																																																					
0	IPBytes			0	1	1	0	1																																																																																					
1	TargetIP[7:0]																																																																																												
2	TargetIP[15:8]																																																																																												
3	TargetIP[23:16]																																																																																												
4	TargetIP[31:24]																																																																																												
5	TargetIP[39:32]																																																																																												
6	TargetIP[47:40]																																																																																												
7	TargetIP[55:48]																																																																																												
8	TargetIP[63:56]																																																																																												
Dependencies	PacketEn	Generation Scenario	Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX ¹ .																																																																																										
Description	Provides the target for some control flow transfers																																																																																												
Application	<p>Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</p> <p>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.</p>																																																																																												

NOTES:

1. EENTER, EEXIT, ERESUME, AEX would be possible only for 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 35-19. 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 35-19), 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 35-20. 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) Packet

Table 35-21. TIP.PGE Packet Definition

Name	Target IP - Packet Generation Enable (TIP.PGE) Packet																																																																																																	
Packet Format	<table border="1" data-bbox="313 373 1302 747"> <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>									7	6	5	4	3	2	1	0	0	IPBytes			1	0	0	0	1	1	TargetIP[7:0]								2	TargetIP[15:8]								3	TargetIP[23:16]								4	TargetIP[31:24]								5	TargetIP[39:32]								6	TargetIP[47:40]								7	TargetIP[55:48]								8	TargetIP[63:56]							
	7	6	5	4	3	2	1	0																																																																																										
0	IPBytes			1	0	0	0	1																																																																																										
1	TargetIP[7:0]																																																																																																	
2	TargetIP[15:8]																																																																																																	
3	TargetIP[23:16]																																																																																																	
4	TargetIP[31:24]																																																																																																	
5	TargetIP[39:32]																																																																																																	
6	TargetIP[47:40]																																																																																																	
7	TargetIP[55:48]																																																																																																	
8	TargetIP[63:56]																																																																																																	
Dependencies	PacketEn transitions to 1	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn																																																																																															
Description	<p>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:</p> <ul style="list-style-type: none"> ▪ 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. 																																																																																																	
Application	TIP.PGE packets bind to the instruction at the IP given in the payload.																																																																																																	

35.4.2.5 Packet Generation Disable (TIP.PGD) Packet

Table 35-22. TIP.PGD Packet Definition

Name	Target IP - Packet Generation Disable (TIP.PGD) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			0	0	0	0	1
	1	TargetIP[7:0]							
	2	TargetIP[15:8]							
	3	TargetIP[23:16]							
	4	TargetIP[31:24]							
	5	TargetIP[39:32]							
	6	TargetIP[47:40]							
	7	TargetIP[55:48]							
	8	TargetIP[63:56]							
Dependencies	PacketEn transitions to 0	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn						
Description	<p>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.</p> <p>PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:</p> <ul style="list-style-type: none"> ▪ 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 cleared 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. <p>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 TNT 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.</p>								
Application	<p>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.</p> <p>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.</p> <p>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.</p>								

35.4.2.6 Flow Update (FUP) Packet

Table 35-23. FUP Packet Definition

Name	Flow Update (FUP) Packet																																																																																												
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table>				7	6	5	4	3	2	1	0	0	IPBytes			1	1	1	0	1	1	IP[7:0]								2	IP[15:8]								3	IP[23:16]								4	IP[31:24]								5	IP[39:32]								6	IP[47:40]								7	IP[55:48]								8	IP[63:56]							
	7	6	5	4	3	2	1	0																																																																																					
0	IPBytes			1	1	1	0	1																																																																																					
1	IP[7:0]																																																																																												
2	IP[15:8]																																																																																												
3	IP[23:16]																																																																																												
4	IP[31:24]																																																																																												
5	IP[39:32]																																																																																												
6	IP[47:40]																																																																																												
7	IP[55:48]																																																																																												
8	IP[63:56]																																																																																												
Dependencies	TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 35.2.4 for details.)	Generation Scenario	Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit, #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, EENTER, EEXIT, ERESUME, EEE, AEX, ¹ INTO, INT1, INT3, INT <i>n</i> , a WRMSR that disables packet generation.																																																																																										
Description	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.																																																																																												
Application	<p>FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.</p> <p>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.</p> <p>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) packet will provide the destination IP. Other packets may be included in the compound event between the FUP and TIP.</p>																																																																																												

NOTES:

1. EENTER, EEXIT, ERESUME, EEE, AEX apply only 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 35-24 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

Table 35-24. 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 35-25. PIP Packet Definition

Name	Paging Information (PIP) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	1	0
	1	0	1	0	0	0	0	1	1
	2	CR3[11:5] or 0							RSVD/NR
	3	CR3[19:12]							
	4	CR3[27:20]							
	5	CR3[35:28]							
	6	CR3[43:36]							
	7	CR3[51:44]							
Dependencies	TriggerEn && ContextEn && IA32_RTIT_CTL.OS			Generation Scenario		MOV CR3, Task switch, INIT, SIPI, PSB+, VM exit, VM entry			
Description	<p>The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other paging modes (32-bit and 4-level paging¹), these bits are 0.</p> <p>This packet holds the CR3 address value. It will be generated on operations that modify CR3:</p> <ul style="list-style-type: none"> ▪ MOV CR3 operation ▪ Task Switch ▪ INIT and SIPI ▪ VM exit, if “conceal VMX from PT” VM-exit control is 0 (see Section 35.5.1) ▪ VM entry, if “conceal VMX from PT” VM-entry control is 0 <p>PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section 35.2.8.3 for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.</p> <p>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.</p> <p>The PIP packet contains the new CR3 value when CR3 is written.</p> <p>PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX from PT” VM-execution control is 0 (see Section 35.5.1). All other PIPs clear the NR bit.</p>								
Application	<p>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:</p> <ol style="list-style-type: none"> 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. <p>For examples of the packets generated by these flows, see Section 35.7.</p>								

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 35-26. 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 35-27. MODE.Exec Packet Definition

Name	MODE.Exec Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table>				7	6	5	4	3	2	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	CS.D	(CS.L & LMA)
	7	6	5	4	3	2	1	0																						
0	1	0	0	1	1	0	0	1																						
1	0	0	0	0	0	0	CS.D	(CS.L & LMA)																						
Dependencies	PacketEn	Generation Scenario	Far branch, interrupt, exception, VM exit, and VM entry, 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.																											
Description	<p>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.</p> <table border="1"> <thead> <tr> <th>CS.D</th> <th>(CS.L & IA32_EFER.LMA)</th> <th>Addressing Mode</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>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.</p>			CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode	1	1	N/A	0	1	64-bit mode	1	0	32-bit mode	0	0	16-bit mode												
CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode																												
1	1	N/A																												
0	1	64-bit mode																												
1	0	32-bit mode																												
0	0	16-bit mode																												
Application	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.																													

MODE.TSX Packet

Table 35-28. MODE.TSX Packet Definition

Name	MODE.TSX Packet																																			
Packet Format	<table border="1" style="width:100%; text-align:center;"> <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>1</td> <td>0</td> <td>0</td> <td>0</td> <td>TXAbort</td> <td>InTX</td> </tr> </table>										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
		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																												
Dependencies	TriggerEn and ContextEn	Generation Scenario	XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if InTX changes, Asynchronous TSX Abort, PSB+																																	
Description	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.																																			
	<table border="1" style="width:100%; text-align:center;"> <thead> <tr> <th>TXAbort</th> <th>InTX</th> <th>Implication</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>N/A</td> </tr> <tr> <td>0</td> <td>1</td> <td>Transaction begins, or executing transactionally</td> </tr> <tr> <td>1</td> <td>0</td> <td>Transaction aborted</td> </tr> <tr> <td>0</td> <td>0</td> <td>Transaction committed, or not executing transactionally</td> </tr> </tbody> </table>									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												
	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																																		
Application																																				
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.																																				

35.4.2.9 TraceStop Packet

Table 35-29. TraceStop Packet Definition

Name	TraceStop Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <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> <th>1</th> <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> </tbody> </table>				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	1
	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	1																						
Dependencies	TriggerEn & ContextEn	Generation Scenario	Taken branch with target in TraceStop IP region, MOV CR3 in TraceStop IP region, or WRMSR that sets TraceEn in TraceStop IP region.																											
Description	<p>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.</p> <p>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.</p>																													
Application	<p>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).</p> <p>If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.</p>																													

35.4.2.10 Core:Bus Ratio (CBR) Packet

Table 35-30. CBR Packet Definition

Name	Core:Bus Ratio (CBR) Packet																																															
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <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> <th>1</th> <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> <th>2</th> <td colspan="8">Core:Bus Ratio</td> </tr> <tr> <th>3</th> <td colspan="8">Reserved</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	2	Core:Bus Ratio								3	Reserved							
	7	6	5	4	3	2	1	0																																								
0	0	0	0	0	0	0	1	0																																								
1	0	0	0	0	0	0	1	1																																								
2	Core:Bus Ratio																																															
3	Reserved																																															
Dependencies	TriggerEn	Generation Scenario	After any frequency change, on C-state wake up, PSB+, and after enabling trace packet generation.																																													
Description	Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.																																															
Application	The CBR packet indicates the point in the trace when a frequency transition has occurred. On some implementations, software execution will continue during transitions to a new frequency, while on others software execution ceases during frequency transitions. There is not a precise IP provided, to which to bind the CBR packet.																																															

35.4.2.11 Timestamp Counter (TSC) Packet

Table 35-31. TSC Packet Definition

Name	Timestamp Counter (TSC) Packet																																																																																			
Packet Format	<table border="1" data-bbox="337 378 1325 709"> <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>				7	6	5	4	3	2	1	0	0	0	0	0	1	1	0	0	1	1	SW TSC[7:0]								2	SW TSC[15:8]								3	SW TSC[23:16]								4	SW TSC[31:24]								5	SW TSC[39:32]								6	SW TSC[47:40]								7	SW TSC[55:48]							
	7	6	5	4	3	2	1	0																																																																												
0	0	0	0	1	1	0	0	1																																																																												
1	SW TSC[7:0]																																																																																			
2	SW TSC[15:8]																																																																																			
3	SW TSC[23:16]																																																																																			
4	SW TSC[31:24]																																																																																			
5	SW TSC[39:32]																																																																																			
6	SW TSC[47:40]																																																																																			
7	SW TSC[55:48]																																																																																			
Dependencies	IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	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.																																																																																	
Description	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.																																																																																			
Application	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.																																																																																			

35.4.2.12 Mini Time Counter (MTC) Packet

Table 35-32. MTC Packet Definition

Name	Mini time Counter (MTC) Packet																													
Packet Format	<table border="1"> <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>				7	6	5	4	3	2	1	0	0	0	1	0	1	1	0	0	1	1	CTC[N+7:N]							
	7	6	5	4	3	2	1	0																						
0	0	1	0	1	1	0	0	1																						
1	CTC[N+7:N]																													
Dependencies	IA32_RTIT_CTL.MTCEn && TriggerEn	Generation Scenario	Periodic, based on the core crystal clock, or Always Running Timer (ART).																											
Description	<p>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 $(ART \gg N) \& FFH$. 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.</p> <p>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.</p> <p>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.</p>																													
Application	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.																													

35.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 35-33. TMA Packet Definition

Name	TSC/MTC Alignment (TMA) Packet																																																																										
Packet Format	<table border="1"> <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>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	1	1	2	CTC[7:0]								3	CTC[15:8]								4	Reserved							0	5	FastCounter[7:0]								6	Reserved							FC[8]
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	0	1	1	1	0	0	1	1																																																																			
2	CTC[7:0]																																																																										
3	CTC[15:8]																																																																										
4	Reserved							0																																																																			
5	FastCounter[7:0]																																																																										
6	Reserved							FC[8]																																																																			
Dependencies	IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	Sent with any TSC packet.																																																																								
Description	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.																																																																										
Application	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.																																																																										

35.4.2.14 Cycle Count (CYC) Packet

Table 35-34. Cycle Count Packet Definition

Name	Cycle Count (CYC) Packet																																															
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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="6">Cycle Counter[11:5]</td> <td colspan="2">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> </tbody> </table>				7	6	5	4	3	2	1	0	0	Cycle Counter[4:0]					Exp	1	1	1	Cycle Counter[11:5]						Exp		2	Cycle Counter[18:12]							Exp (if Exp = 1 in the previous byte)							
	7	6	5	4	3	2	1	0																																								
0	Cycle Counter[4:0]					Exp	1	1																																								
1	Cycle Counter[11:5]						Exp																																									
2	Cycle Counter[18:12]							Exp																																								
...	... (if Exp = 1 in the previous byte)																																															
Dependencies	IA32_RTIT_CTL.CYCEn && TriggerEn	Generation Scenario	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.																																													
Description	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>																																															
Application	<p>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.</p> <p>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.</p>																																															

35.4.2.15 VMCS Packet

Table 35-35. VMCS Packet Definition

Name	VMCS Packet																																																																										
Packet Format	<table border="1"> <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 pointer [19:12]</td> </tr> <tr> <td>3</td> <td colspan="8">VMCS pointer [27:20]</td> </tr> <tr> <td>4</td> <td colspan="8">VMCS pointer [35:28]</td> </tr> <tr> <td>5</td> <td colspan="8">VMCS pointer [43:36]</td> </tr> <tr> <td>6</td> <td colspan="8">VMCS pointer [51:44]</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	1	0	0	0	2	VMCS pointer [19:12]								3	VMCS pointer [27:20]								4	VMCS pointer [35:28]								5	VMCS pointer [43:36]								6	VMCS pointer [51:44]							
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	1	1	0	0	1	0	0	0																																																																			
2	VMCS pointer [19:12]																																																																										
3	VMCS pointer [27:20]																																																																										
4	VMCS pointer [35:28]																																																																										
5	VMCS pointer [43:36]																																																																										
6	VMCS pointer [51:44]																																																																										
Dependencies	TriggerEn && ContextEn; Also in VMX operation.	Generation Scenario	Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 35.4.2.26).																																																																								
Description	<p>The VMCS packet provides a VMCS pointer for a decoder to determine the transition of code contexts:</p> <ul style="list-style-type: none"> On a successful VMPTRLD (i.e., a VMPTRLD that doesn't fault, fail, or VM exit), the VMCS packet contains the logical processor's VMCS pointer established by VMPTRLD (for subsequent execution of a VM guest context). An SMM VM exit loads the logical processor's VMCS pointer with the SMM-transfer VMCS pointer. If the "conceal VMX from PT" VM-exit control is 0 (see Section 35.5.1), a VMCS packet provides this pointer. See Section 35.6 on tracing inside and outside STM. A VM entry that returns from SMM loads the logical processor's VMCS pointer from a field in the SMM-transfer VMCS. If the "conceal VMX from PT" VM-entry control is 0, a VMCS packet provides this pointer. Whether the VM entry is to VMX root operation or VMX non-root operation is indicated by the PIP.NR bit. <p>A VMCS packet generated before a VMCS pointer has been loaded, or after the VMCS pointer has been cleared will set all 64 bits in the VMCS pointer field.</p> <p>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.</p>																																																																										
Application	<p>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.</p> <p>When a VMCS packet is encountered, a decoder should do the following:</p> <ul style="list-style-type: none"> 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. <p>For examples of the packets generated by these flows, see Section 35.7.</p>																																																																										

35.4.2.16 Overflow (OVF) Packet

Table 35-36. OVF Packet Definition

Name	Overflow (OVF) Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <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> <th>1</th> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </tbody> </table>				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
	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																						
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.																													

35.4.2.17 Packet Stream Boundary (PSB) Packet

Table 35-37. PSB Packet Definition

Name	Packet Stream Boundary (PSB) Packet																																																																																																																																																																
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <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> <th>1</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>2</th> <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> <th>3</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>4</th> <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> <th>5</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>6</th> <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> <th>7</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>8</th> <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> <th>9</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>10</th> <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> <th>11</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>12</th> <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> <th>13</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>14</th> <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> <th>15</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> </tbody> </table>									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
	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 35-37. 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	<p>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.</p> <p>PSB also serves as the leading packet for a set of “status-only” packets collectively known as PSB+ (Section 35.3.7).</p>		
Application	<p>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.</p>		

35.4.2.18 PSBEND Packet

Table 35-38. PSBEND Packet Definition

Name	PSBEND Packet																																		
Packet Format	<table border="1" style="width: 100%; text-align: center;"> <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>1</td> </tr> </table>									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
	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																											
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”.																																		

35.4.2.19 Maintenance (MNT) Packet

Table 35-39. MNT Packet Definition

Name	Maintenance (MNT) Packet																																																																																																																			
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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>1</td> </tr> <tr> <td>2</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>3</td> <td colspan="8">Payload[7:0]</td> </tr> <tr> <td>4</td> <td colspan="8">Payload[15:8]</td> </tr> <tr> <td>5</td> <td colspan="8">Payload[23:16]</td> </tr> <tr> <td>6</td> <td colspan="8">Payload[31:24]</td> </tr> <tr> <td>7</td> <td colspan="8">Payload[39:32]</td> </tr> <tr> <td>8</td> <td colspan="8">Payload[47:40]</td> </tr> <tr> <td>9</td> <td colspan="8">Payload[55:48]</td> </tr> <tr> <td>10</td> <td colspan="8">Payload[63:56]</td> </tr> </tbody> </table>									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]							
	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]																																																																																																																			
Dependencies	TriggerEn	Generation Scenario	Implementation specific.																																																																																																																	
Description	This packet is generated by hardware, the payload meaning is model-specific.																																																																																																																			
Application	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.																																																																																																																			

35.4.2.20 PAD Packet

Table 35-40. PAD Packet Definition

Name	PAD Packet																									
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0
	7	6	5	4	3	2	1	0																		
0	0	0	0	0	0	0	0	0																		
Dependencies	TriggerEn	Generation Scenario	Implementation specific																							
Description	PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.																									
Application	Ignore PAD packets.																									

35.4.2.21 PTWRITE (PTW) Packet

Table 35-41. PTW Packet Definition

Name	PTW Packet																																																																																																										
Packet Format	<table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr style="background-color: #f2f2f2;"> <td style="width: 20px;"></td> <td style="width: 20px;">7</td> <td style="width: 20px;">6</td> <td style="width: 20px;">5</td> <td style="width: 20px;">4</td> <td style="width: 20px;">3</td> <td style="width: 20px;">2</td> <td style="width: 20px;">1</td> <td style="width: 20px;">0</td> </tr> <tr style="background-color: #e0f2e0;"> <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 style="background-color: #e0f2e0;"> <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 style="background-color: #e0f2e0;"> <td>2</td> <td colspan="8">Payload[7:0]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>3</td> <td colspan="8">Payload[15:8]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>4</td> <td colspan="8">Payload[23:16]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>5</td> <td colspan="8">Payload[31:24]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>6</td> <td colspan="8">Payload[39:32]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>7</td> <td colspan="8">Payload[47:40]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>8</td> <td colspan="8">Payload[55:48]</td> </tr> <tr style="background-color: #e0f2e0;"> <td>9</td> <td colspan="8">Payload[63:56]</td> </tr> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	PayloadBytes		1	0	0	1	0	2	Payload[7:0]								3	Payload[15:8]								4	Payload[23:16]								5	Payload[31:24]								6	Payload[39:32]								7	Payload[47:40]								8	Payload[55:48]								9	Payload[63:56]							
	7	6	5	4	3	2	1	0																																																																																																			
0	0	0	0	0	0	0	1	0																																																																																																			
1	IP	PayloadBytes		1	0	0	1	0																																																																																																			
2	Payload[7:0]																																																																																																										
3	Payload[15:8]																																																																																																										
4	Payload[23:16]																																																																																																										
5	Payload[31:24]																																																																																																										
6	Payload[39:32]																																																																																																										
7	Payload[47:40]																																																																																																										
8	Payload[55:48]																																																																																																										
9	Payload[63:56]																																																																																																										
	<p>The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="width: 30%;">PayloadBytes</th> <th style="width: 70%;">Bytes of Payload</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</p>								PayloadBytes	Bytes of Payload	'00	4	'01	8	'10	Reserved	'11	Reserved																																																																																									
PayloadBytes	Bytes of Payload																																																																																																										
'00	4																																																																																																										
'01	8																																																																																																										
'10	Reserved																																																																																																										
'11	Reserved																																																																																																										
Dependencies	TriggerEn & ContextEn & FilterEn & PTWEn	Generation Scenario	PTWRITE Instruction																																																																																																								
Description	<p>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.</p>																																																																																																										
Application	<p>Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.</p>																																																																																																										

35.4.2.22 Execution Stop (EXSTOP) Packet

Table 35-42. EXSTOP Packet Definition

Name	EXSTOP Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>IP bit indicates if a FUP will follow.</p>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	IP	1	1	0	0	0	1	0																						
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	<p>C-state entry, P-state change, or other processor clock power-down. Includes :</p> <ul style="list-style-type: none"> ▪ Entry to C-state deeper than C0.0 ▪ TM1/2 ▪ STPCLK# ▪ Frequency change due to IA32_CLOCK_MODULATION, Turbo 																											
Description	<p>This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes.</p> <p>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.</p> <p>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.</p>																													
Application	<p>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.</p>																													

35.4.2.23 MWAIT Packet

Table 35-43. MWAIT Packet Definition

Name	MWAIT Packet																																																																																																					
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table>				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	0	2	MWAIT Hints[7:0]								3	Reserved								4	Reserved								5	Reserved								6	Reserved						EXT[1:0]		7	Reserved								8	Reserved								9	Reserved							
	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	0																																																																																														
2	MWAIT Hints[7:0]																																																																																																					
3	Reserved																																																																																																					
4	Reserved																																																																																																					
5	Reserved																																																																																																					
6	Reserved						EXT[1:0]																																																																																															
7	Reserved																																																																																																					
8	Reserved																																																																																																					
9	Reserved																																																																																																					
Dependencies	TriggerEn & PwrEvtEn & ContextEn	Generation Scenario	MWAIT, UMWAIT, or TPAUSE instructions, or I/O redirection to MWAIT, that complete without fault or VMexit.																																																																																																			
Description	<p>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. For UMWAIT and TPAUSE, the EXT field holds the input register value that determines the optimized state requested.</p> <p>For entry to some highly optimized C0 sub-C-states, such as C0.1, no MWAIT packet is generated.</p> <p>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.</p>																																																																																																					
Application	The binding for the upcoming EXSTOP packet also applies to the MWAIT packet. See Section 35.4.2.22.																																																																																																					

35.4.2.24 Power Entry (PWRE) Packet

Table 35-44. PWRE Packet Definition

Name	PWRE Packet																																															
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <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> </tbody> </table>				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	0	2	HW	Reserved							3	Resolved Thread C-State				Resolved Thread Sub C-State			
	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	0																																								
2	HW	Reserved																																														
3	Resolved Thread C-State				Resolved Thread Sub C-State																																											
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition to a C-state deeper than C0.0.																																													
Description	<p>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 C0.0.</p> <p>For entry to some highly optimized C0 sub-C-states, such as C0.1, no PWRE packet is generated.</p> <p>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.</p> <p>If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, UMWAIT, TPAUSE, HLT, or shutdown), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 14.5, "Hardware Duty Cycling (HDC)" in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B</i>) is an example of such a case.</p>																																															
Application	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.																																															

35.4.2.25 Power Exit (PWRX) Packet

Table 35-45. PWRX Packet Definition

Name	PWRX Packet																																																																										
Packet Format	<table border="1"> <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>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1	0	2	Last Core C-State				Deepest Core C-State				3	Reserved				Wake Reason				4	Reserved								5	Reserved								6	Reserved							
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	1	0	1	0	0	0	1	0																																																																			
2	Last Core C-State				Deepest Core C-State																																																																						
3	Reserved				Wake Reason																																																																						
4	Reserved																																																																										
5	Reserved																																																																										
6	Reserved																																																																										
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition from a C-state deeper than C0.0 to C0.																																																																								
Description	<p>Indicates processor return to thread C0 from a C-state deeper than C0.0. For return from some highly optimized C0 sub-C-states, such as C0.1, no PWRX packet is generated. 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 <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B</i>. 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:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Field</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Interrupt</td> <td>Wake due to external interrupt received.</td> </tr> <tr> <td>1</td> <td>Timer Deadline</td> <td>Wake due to timer expiration, such as UMWAIT/TPAUSE TSC-quanta.</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> </tbody> </table>			Bit	Field	Meaning	0	Interrupt	Wake due to external interrupt received.	1	Timer Deadline	Wake due to timer expiration, such as UMWAIT/TPAUSE TSC-quanta.	2	Store to Monitored Address	Wake due to store to monitored address.	3	Hw Wake	Wake due to hardware autonomous condition, such as HDC.																																																									
Bit	Field	Meaning																																																																									
0	Interrupt	Wake due to external interrupt received.																																																																									
1	Timer Deadline	Wake due to timer expiration, such as UMWAIT/TPAUSE TSC-quanta.																																																																									
2	Store to Monitored Address	Wake due to store to monitored address.																																																																									
3	Hw Wake	Wake due to hardware autonomous condition, such as HDC.																																																																									
Application	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.																																																																										

35.4.2.26 Block Begin Packet (BBP)

Table 35-46. Block Begin Packet Definition

Name	BBP																																						
Packet Format	<table border="1"> <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>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td>SZ</td> <td colspan="2">Reserved</td> <td colspan="5">Type[4:0]</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	2	SZ	Reserved		Type[4:0]				
	7	6	5	4	3	2	1	0																															
0	0	0	0	0	0	0	1	0																															
1	0	1	1	0	0	0	1	1																															
2	SZ	Reserved		Type[4:0]																																			
Dependencies	TriggerEn	Generation Scenario	PEBS event, if IA32_PEBS_ENABLE.OUTPUT=1.																																				
Description	<p>This packet indicates the beginning of a block of packets which are collectively tied to a single event or instruction. The size of the block item payloads within this block is provided by the Size (SZ) bit: SZ=0: 8-byte block items SZ=1: 4-byte block items The meaning of the BIP payloads is provided by the Type field:</p> <table border="1"> <thead> <tr> <th>BBP.Type</th> <th>Block name</th> </tr> </thead> <tbody> <tr> <td>0x00</td> <td>Reserved</td> </tr> <tr> <td>0x01</td> <td>General-Purpose Registers</td> </tr> <tr> <td>0x02..0x03</td> <td>Reserved</td> </tr> <tr> <td>0x04</td> <td>PEBS Basic</td> </tr> <tr> <td>0x05</td> <td>PEBS Memory</td> </tr> <tr> <td>0x06..0x07</td> <td>Reserved</td> </tr> <tr> <td>0x08</td> <td>LBR Block 0</td> </tr> <tr> <td>0x09</td> <td>LBR Block 1</td> </tr> <tr> <td>0x0A</td> <td>LBR Block 2</td> </tr> <tr> <td>0x0B..0x0F</td> <td>Reserved</td> </tr> <tr> <td>0x10</td> <td>XMM Registers</td> </tr> <tr> <td>0x11..0x1F</td> <td>Reserved</td> </tr> </tbody> </table>			BBP.Type	Block name	0x00	Reserved	0x01	General-Purpose Registers	0x02..0x03	Reserved	0x04	PEBS Basic	0x05	PEBS Memory	0x06..0x07	Reserved	0x08	LBR Block 0	0x09	LBR Block 1	0x0A	LBR Block 2	0x0B..0x0F	Reserved	0x10	XMM Registers	0x11..0x1F	Reserved										
BBP.Type	Block name																																						
0x00	Reserved																																						
0x01	General-Purpose Registers																																						
0x02..0x03	Reserved																																						
0x04	PEBS Basic																																						
0x05	PEBS Memory																																						
0x06..0x07	Reserved																																						
0x08	LBR Block 0																																						
0x09	LBR Block 1																																						
0x0A	LBR Block 2																																						
0x0B..0x0F	Reserved																																						
0x10	XMM Registers																																						
0x11..0x1F	Reserved																																						
Application	A BBP will always be followed by a Block End Packet (BEP), and when the block is generated while ContextEn=1 that BEP will have IP=1 and be followed by a FUP that provides the IP to which the block should be bound. Note that, in addition to BEP, a block can be terminated by a BBP (indicating the start of a new block) or an OVF packet.																																						

35.4.2.27 Block Item Packet (BIP)

Table 35-47. Block Item Packet Definition

Name	BIP																																																																																																																																																		
Packet Format	<p>If the preceding BBP.SZ=0:</p> <table border="1" style="margin-left: 20px;"> <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">ID[5:0]</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td colspan="8">Payload[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">Payload[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">Payload[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">Payload[31:24]</td> </tr> <tr> <td>5</td> <td colspan="8">Payload[39:32]</td> </tr> <tr> <td>6</td> <td colspan="8">Payload[47:40]</td> </tr> <tr> <td>7</td> <td colspan="8">Payload[55:48]</td> </tr> <tr> <td>8</td> <td colspan="8">Payload[63:56]</td> </tr> </table> <p>If the preceding BBP.SZ=1:</p> <table border="1" style="margin-left: 20px;"> <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">ID[5:0]</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td colspan="8">Payload[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">Payload[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">Payload[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">Payload[31:24]</td> </tr> </table>				7	6	5	4	3	2	1	0	0	ID[5:0]					1	0	0	1	Payload[7:0]								2	Payload[15:8]								3	Payload[23:16]								4	Payload[31:24]								5	Payload[39:32]								6	Payload[47:40]								7	Payload[55:48]								8	Payload[63:56]									7	6	5	4	3	2	1	0	0	ID[5:0]					1	0	0	1	Payload[7:0]								2	Payload[15:8]								3	Payload[23:16]								4	Payload[31:24]							
	7	6	5	4	3	2	1	0																																																																																																																																											
0	ID[5:0]					1	0	0																																																																																																																																											
1	Payload[7:0]																																																																																																																																																		
2	Payload[15:8]																																																																																																																																																		
3	Payload[23:16]																																																																																																																																																		
4	Payload[31:24]																																																																																																																																																		
5	Payload[39:32]																																																																																																																																																		
6	Payload[47:40]																																																																																																																																																		
7	Payload[55:48]																																																																																																																																																		
8	Payload[63:56]																																																																																																																																																		
	7	6	5	4	3	2	1	0																																																																																																																																											
0	ID[5:0]					1	0	0																																																																																																																																											
1	Payload[7:0]																																																																																																																																																		
2	Payload[15:8]																																																																																																																																																		
3	Payload[23:16]																																																																																																																																																		
4	Payload[31:24]																																																																																																																																																		
Dependencies	TriggerEn	Generation Scenario	See BBP.																																																																																																																																																
Description	<p>The size of the BIP payload is determined by the Size field in the preceding BBP packet. The BIP header provides the ID value that, when combined with the Type field from the preceding BBP, uniquely identifies the state value held in the BIP payload. See Table 35-48 below for the complete list.</p>																																																																																																																																																		
Application	See BBP.																																																																																																																																																		

BIP State Value Encodings

The table below provides the encoding values for all defined block items. State items that are larger than 8 bytes, such as XMM register values, are broken into multiple 8-byte components. BIP packets with Size=1 (4 byte payload) will provide only the lower 4 bytes of the associated state value.

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
General-Purpose Registers		
0x01	0x00	R/EFLAGS
0x01	0x01	R/EIP
0x01	0x02	R/EAX
0x01	0x03	R/ECX

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
0x01	0x04	R/EDX
0x01	0x05	R/EBX
0x01	0x06	R/ESP
0x01	0x07	R/EBP
0x01	0x08	R/ESI
0x01	0x09	R/EDI
0x01	0x0A	R8
0x01	0x0B	R9
0x01	0x0C	R10
0x01	0x0D	R11
0x01	0x0E	R12
0x01	0x0F	R13
0x01	0x10	R14
0x01	0x11	R15
PEBS Basic Info (Section 18.9.2.2.1)		
0x04	0x00	Instruction Pointer
0x04	0x01	Applicable Counters
0x04	0x02	Timestamp
PEBS Memory Info (Section 18.9.2.2.2)		
0x05	0x00	MemAccessAddress
0x05	0x01	MemAuxInfo
0x05	0x02	MemAccessLatency
0x05	0x03	TSXAuxInfo
LBR_0		
0x08	0x00	LBR[TOS-0]_FROM_IP
0x08	0x01	LBR[TOS-0]_TO_IP
0x08	0x02	LBR[TOS-0]_INFO
0x08	0x03	LBR[TOS-1]_FROM_IP
0x08	0x04	LBR[TOS-1]_TO_IP
0x08	0x05	LBR[TOS-1]_INFO
0x08	0x06	LBR[TOS-2]_FROM_IP
0x08	0x07	LBR[TOS-2]_TO_IP
0x08	0x08	LBR[TOS-2]_INFO
0x08	0x09	LBR[TOS-3]_FROM_IP
0x08	0x0A	LBR[TOS-3]_TO_IP
0x08	0x0B	LBR[TOS-3]_INFO
0x08	0x0C	LBR[TOS-4]_FROM_IP
0x08	0x0D	LBR[TOS-4]_TO_IP

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
0x08	0x0E	LBR[TOS-4]_INFO
0x08	0x0F	LBR[TOS-5]_FROM_IP
0x08	0x10	LBR[TOS-5]_TO_IP
0x08	0x11	LBR[TOS-5]_INFO
0x08	0x12	LBR[TOS-6]_FROM_IP
0x08	0x13	LBR[TOS-6]_TO_IP
0x08	0x14	LBR[TOS-6]_INFO
0x08	0x15	LBR[TOS-7]_FROM_IP
0x08	0x16	LBR[TOS-7]_TO_IP
0x08	0x17	LBR[TOS-7]_INFO
0x08	0x18	LBR[TOS-8]_FROM_IP
0x08	0x19	LBR[TOS-8]_TO_IP
0x08	0x1A	LBR[TOS-8]_INFO
0x08	0x1B	LBR[TOS-9]_FROM_IP
0x08	0x1C	LBR[TOS-9]_TO_IP
0x08	0x1D	LBR[TOS-9]_INFO
0x08	0x1E	LBR[TOS-10]_FROM_IP
0x08	0x1F	LBR[TOS-10]_TO_IP
LBR_1		
0x09	0x00	LBR[TOS-10]_INFO
0x09	0x01	LBR[TOS-11]_FROM_IP
0x09	0x02	LBR[TOS-11]_TO_IP
0x09	0x03	LBR[TOS-11]_INFO
0x09	0x04	LBR[TOS-12]_FROM_IP
0x09	0x05	LBR[TOS-12]_TO_IP
0x09	0x06	LBR[TOS-12]_INFO
0x09	0x07	LBR[TOS-13]_FROM_IP
0x09	0x08	LBR[TOS-13]_TO_IP
0x09	0x09	LBR[TOS-13]_INFO
0x09	0x0A	LBR[TOS-14]_FROM_IP
0x09	0x0B	LBR[TOS-14]_TO_IP
0x09	0x0C	LBR[TOS-14]_INFO
0x09	0x0D	LBR[TOS-15]_FROM_IP
0x09	0x0E	LBR[TOS-15]_TO_IP
0x09	0x0F	LBR[TOS-15]_INFO
0x09	0x10	LBR[TOS-16]_FROM_IP
0x09	0x11	LBR[TOS-16]_TO_IP
0x09	0x12	LBR[TOS-16]_INFO

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
0x09	0x13	LBR[TOS-17]_FROM_IP
0x09	0x14	LBR[TOS-17]_TO_IP
0x09	0x15	LBR[TOS-17]_INFO
0x09	0x16	LBR[TOS-18]_FROM_IP
0x09	0x17	LBR[TOS-18]_TO_IP
0x09	0x18	LBR[TOS-18]_INFO
0x09	0x19	LBR[TOS-19]_FROM_IP
0x09	0x1A	LBR[TOS-19]_TO_IP
0x09	0x1B	LBR[TOS-19]_INFO
0x09	0x1C	LBR[TOS-20]_FROM_IP
0x09	0x1D	LBR[TOS-20]_TO_IP
0x09	0x1E	LBR[TOS-20]_INFO
0x09	0x1F	LBR[TOS-21]_FROM_IP
LBR_2		
0x0A	0x00	LBR[TOS-21]_TO_IP
0x0A	0x01	LBR[TOS-21]_INFO
0x0A	0x02	LBR[TOS-22]_FROM_IP
0x0A	0x03	LBR[TOS-22]_TO_IP
0x0A	0x04	LBR[TOS-22]_INFO
0x0A	0x05	LBR[TOS-23]_FROM_IP
0x0A	0x06	LBR[TOS-23]_TO_IP
0x0A	0x07	LBR[TOS-23]_INFO
0x0A	0x08	LBR[TOS-24]_FROM_IP
0x0A	0x09	LBR[TOS-24]_TO_IP
0x0A	0x0A	LBR[TOS-24]_INFO
0x0A	0x0B	LBR[TOS-25]_FROM_IP
0x0A	0x0C	LBR[TOS-25]_TO_IP
0x0A	0x0D	LBR[TOS-25]_INFO
0x0A	0x0E	LBR[TOS-26]_FROM_IP
0x0A	0x0F	LBR[TOS-26]_TO_IP
0x0A	0x10	LBR[TOS-26]_INFO
0x0A	0x11	LBR[TOS-27]_FROM_IP
0x0A	0x12	LBR[TOS-27]_TO_IP
0x0A	0x13	LBR[TOS-27]_INFO
0x0A	0x14	LBR[TOS-28]_FROM_IP
0x0A	0x15	LBR[TOS-28]_TO_IP
0x0A	0x16	LBR[TOS-28]_INFO
0x0A	0x17	LBR[TOS-29]_FROM_IP

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
0x0A	0x18	LBR[TOS-29]_TO_IP
0x0A	0x19	LBR[TOS-29]_INFO
0x0A	0x1A	LBR[TOS-30]_FROM_IP
0x0A	0x1B	LBR[TOS-30]_TO_IP
0x0A	0x1C	LBR[TOS-30]_INFO
0x0A	0x1D	LBR[TOS-31]_FROM_IP
0x0A	0x1E	LBR[TOS-31]_TO_IP
0x0A	0x1F	LBR[TOS-31]_INFO
XMM Registers		
0x10	0x00	XMM0_Q0
0x10	0x01	XMM0_Q1
0x10	0x02	XMM1_Q0
0x10	0x03	XMM1_Q1
0x10	0x04	XMM2_Q0
0x10	0x05	XMM2_Q1
0x10	0x06	XMM3_Q0
0x10	0x07	XMM3_Q1
0x10	0x08	XMM4_Q0
0x10	0x09	XMM4_Q1
0x10	0x0A	XMM5_Q0
0x10	0x0B	XMM5_Q1
0x10	0x0C	XMM6_Q0
0x10	0x0D	XMM6_Q1
0x10	0x0E	XMM7_Q0
0x10	0x0F	XMM7_Q1
0x10	0x10	XMM8_Q0
0x10	0x11	XMM8_Q1
0x10	0x12	XMM9_Q0
0x10	0x13	XMM9_Q1
0x10	0x14	XMM10_Q0
0x10	0x15	XMM10_Q1
0x10	0x16	XMM11_Q0
0x10	0x17	XMM11_Q1
0x10	0x18	XMM12_Q0
0x10	0x19	XMM12_Q1
0x10	0x1A	XMM13_Q0
0x10	0x1B	XMM13_Q1
0x10	0x1C	XMM14_Q0

Table 35-48. BIP Encodings

BBP.Type	BIP.ID	State Value
0x10	0x1D	XMM14_Q1
0x10	0x1E	XMM15_Q0
0x10	0x1F	XMM15_Q1

35.4.2.28 Block End Packet (BEP)

Table 35-49. Block End Packet Definition

Name	BEP																																		
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <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> <th>1</th> <td>IP</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </tbody> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	0	1	1	0	0	1	1
	7	6	5	4	3	2	1	0																											
0	0	0	0	0	0	0	1	0																											
1	IP	0	1	1	0	0	1	1																											
Dependencies	TriggerEn	Generation Scenario	See BBP.																																
Description	Indicates the end of a packet block. The IP bit indicates if a FUP will follow, and will be set if ContextEn=1.																																		
Application	The block, from initial BBP to the BEP, binds to the FUP IP, if IP=1, and consumes the FUP.																																		

35.5 TRACING IN VMX OPERATION

On processors that IA32_VMX_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. The VMM can configure specific VMX controls to control what virtualization-specific data is included within the trace packets (see Section 35.5.1 for details). The VMM can also configure the VMCS to limit tracing to non-root operation, or to trace across both root and non-root operation. The VMCS controls exist to simplify virtualization of Intel PT for guest use, including the "Clear IA32_RTIT_CTL" exit control (See Section 24.7.1), "Load IA32_RTIT_CTL" entry control (See Section 24.8.1), and "Intel PT uses guest physical addresses" execution control (See Section 25.5.3).

For older processors that do not support these VMCS controls, the MSR-load areas used by VMX transitions can be employed by the VMM to restrict tracing to the desired context. See Section 35.5.2 for details. Tracing with SMM Transfer Monitor is described in Section 35.6.

35.5.1 VMX-Specific Packets and VMCS Controls

In all of the usages of VMX and Intel PT, a decoder in the host or VMM context can identify the occurrences of VMX transitions with the aid of VMX-specific packets. There are two kinds of packets relevant to VMX:

- **VMCS packet.** The VMX transitions of individual VMs can be distinguished by a decoder using the VMCS-pointer field in a VMCS packet. A VMCS packet is sent on a successful execution of VMPTRLD, and its VMCS-pointer field stores the VMCS pointer loaded by that execution. See Section 35.4.2.15 for details.
- **The NR (non-root) bit in a PIP packet.** Normally, the NR bit is set in any PIP packet generated in VMX non-root operation. In addition, PIP packets are generated with each VM entry and VM exit. 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.

There are VMX controls that a VMM can set to conceal some of this VMX-specific information (by suppressing its recording) and thereby prevent it from leaking across virtualization boundaries. There is one of these controls (each of which is called "conceal VMX from PT") of each type of VMX control.

Table 35-50. VMX Controls For Intel Processor Trace

Type of VMX Control	Bit Position ¹	Value	Behavior
Secondary processor-based VM-execution control	19	0	Each PIP generated in VM non-root operation will set the 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	Each PIP generated in VMX non-root operation will clear the NR bit. PSB+ in VMX non-root operation will not include the VMCS packet.
VM-exit control	24	0	Each VM exit generates a PIP in which the NR bit is clear. In addition, SMM VM exits generate VMCS packets.
		1	VM exits do not generate PIPs, and no VMCS packets are generated on SMM VM exits.
VM-entry control	17	0	Each VM entry generates a PIP in which the NR bit is set (except VM entries that return from SMM to VMX root operation). In addition, VM entries that return from SMM generate VMCS packets.
		1	VM entries do not generate PIPs, and no VMCS packets are generated on VM entries that return from SMM.

NOTES:

1. These are the positions of the control bits in the relevant VMX control fields.

The 0-settings of these VMX controls enable all VMX-specific packet information. The scenarios that would use these default settings also do not require the VMM to use VMX MSR-load areas to enable and disable trace-packet generation across VMX transitions.

If IA32_VMX_MISC[bit 14] reports 0, the 1-settings of the VMX controls in Table 35-50 are not supported, and VM entry will fail on any attempt to set them.

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 across VMX transitions.

For tracing scenarios that collect packets only within VMX root operation or only within VMX non-root operation, the VMM can toggle IA32_RTIT_CTL.TraceEn on VMX transitions.

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 relevant VMX controls clear to allow VMX-specific packets to provide information across VMX transitions.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 35-51.

Table 35-51. 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 RIP loaded 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 VMX controls that suppress packet generation are cleared, a VMCS packet will be included in all PSB+ for this usage scenario. Additionally, VMPTRLD will generate such a packet. 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] as 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 VMX controls (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 Guest-Only Tracing

A VMM can configure trace-packet generation while in VMX non-root operation for guests executing normally. This is accomplished by utilizing **VMCS controls to manipulate the guest IA32_RTIT_CTL value on VMX transitions. For older processors that do not support these VMCS controls, a VMM can use the VMX MSR-load areas on VM exits** (see Section 24.7.2, "VM-Exit Controls for MSRs") and VM entries (see Section 24.8.2, "VM-Entry Controls for MSRs") to limit trace-packet generation to the guest environment.

For this usage, VM-entry is programmed to enable trace packet generation, while VM-exit is programmed to clear IA32_RTIT_CTL.TraceEn so as 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 to 1 all the VMX controls enumerated in Table 35-50.

35.5.2.3 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.

35.5.2.4 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.

35.5.2.5 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 35-52. Packets on a Failed VM Entry

Usage Model	Entry Configuration	Early Failure (fall through to next IP)	Late Failure (VM-exit like)
System-Wide	No use of "Load IA32_RTIT_CTL" entry control or VM-entry MSR-load area	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	"Load IA32_RTIT_CTL" entry control or VM-entry MSR-load area used to clear TraceEn	TIP (NextIP)	TraceEn 0->1 Packets (See Section 35.2.7.3), TIP(HostIP)
VM Only	"Load IA32_RTIT_CTL" entry control or VM-entry MSR-load area used to set TraceEn	None	None

35.5.2.6 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)

The SMM-transfer monitor (STM) 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 35.5. As a result, on a SMM VM exit resulting from #SMI, TraceEn is neither saved nor cleared by default. 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 35-53 and Table 35-55 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.

The following acronyms are used in the packet examples below:

- CLIP - Current LIP
- NLIP - Next Sequential LIP
- BLIP - Branch Target LIP

In Table 35-53, PktEn is evaluated based on TiggerEn & ContextEn & FilterEn & BranchEn.

Table 35-53. 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
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
5b	MOV to CR3	0	1	1	*PIP.NR=1 if not in root operation and the “conceal VMX from PT” VM-execution control is 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()
5d	MOV to CR3	1	1	1	*PIP.NR=1 if not in root operation and the “conceal VMX from PT” VM-execution control is 0	PIP(NewCR3, NR?)

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
5e	MOV to CR3	1	0	1	*PIP.NR=1 if not in root operation and the "conceal VMX from PT" VM-execution control is 0 *TraceStop if executed in a TraceStop region	PIP(NewCR3, NR?), TIP.PGD(NLIP), TraceStop?
5f	MOV to CR3	0	0	1	TraceStop if executed in a TraceStop region	PIP(NewCR3, NR?), TraceStop?
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
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)
7d	Conditional taken jump or compressed RET that fills up the internal TNT buffer	1	1	1		TNT
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(), TraceStop?
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/SYS*/IRET)	0	0	0		None

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
10b	Far Branch (CALL/JMP/RET/SYS*/IRET)	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 the "conceal VMX from PT" VM-execution control is 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/SYS*/IRET)	1	0	0		TIP.PGD()
10d	Far Branch (CALL/JMP/RET/SYS*/IRET)	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 the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TIP.PGD(BLIP), TraceStop?
10e	Far Branch (CALL/JMP/RET/SYS*/IRET)	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 the "conceal VMX from PT" VM-execution control is 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
10f	Far Branch (CALL/JMP/RET/SYS*/IRET)	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 the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
11a	HW Interrupt	0	0	0		None
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 the "conceal VMX from PT" VM-execution control is 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()

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
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 the “conceal VMX from PT” VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	FUP(NLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
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 the “conceal VMX from PT” VM-execution control is 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(NLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
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 the “conceal VMX from PT” VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
12a	SW Interrupt	0	0	0		None
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 the “conceal VMX from PT” VM-execution control is 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 the “conceal VMX from PT” VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
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 the "conceal VMX from PT" VM-execution control is 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
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 the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
13a	Exception/Fault	0	0	0		None
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 the "conceal VMX from PT" VM-execution control is 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 the "conceal VMX from PT" VM-execution control is 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 the "conceal VMX from PT" VM-execution control is 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)

Table 35-53. 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 the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
14a	SMI (TraceEn cleared)	0	0	D.C.		None
14b	SMI (TraceEn cleared)	1	0	0		FUP(SMRAM.LIP), TIP.PGD()
14c	SMI (TraceEn cleared)	1	1	1		NA
14f	SMI (TraceEn cleared)	1	0	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
15c	RSM, TraceEn restored to 1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15d	RSM (TraceEn=1, goes to shutdown)	1	1	1		None
15e	RSM (TraceEn=1, goes to shutdown)	1	0	0		None
15f	RSM (TraceEn=1, goes to shutdown)	1	0	1		None
16a	VM exit	0	0	1	*PIP if OF=1 and the "conceal VMX from PT" VM-exit control is 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 the "conceal VMX from PT" VM-exit control is 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 the "conceal VMX from PT" VM-exit control is 0;	FUP(VMCSG.LIP), PIP(HostCR3, NR=0?)?, TIP.PGD

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16g	VM exit	1	0	1	*PIP if OF=1 and the “conceal VMX from PT” VM-exit control is 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 the “conceal VMX from PT” VM-exit control is 0; *MODE.Exec if the value is different, since last TIP.PGD	FUP(VMCSG.LIP), PIP(HostCR3, NR=0)?, MODE.Exec, TIP(VMCSH.LIP)
16i	VM exit	0	0	0		None
16j	VM exit, ContextEN 1->0	1	0	0		FUP(VMCSG.LIP), TIP.PGD
17a	VM entry	0	0	0		None
17b	VM entry	0	0	1	*PIP if OF=1 and the “conceal VMX from PT” VM-entry control is 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
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 the “conceal VMX from PT” VM-entry control is 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec?, TIP.PGE(VMCSG.LIP)
17g	VM entry, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1 and the “conceal VMX from PT” VM-entry control is 0;	PIP(GuestCR3, NR=1)?, TIP.PGD
17h	VM entry	1	0	1	*PIP if OF=1 and the “conceal VMX from PT” VM-entry control is 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 the “conceal VMX from PT” VM-entry control is 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec, TIP(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)
20a	EENTER/ERESUME to non-debug enclave	0	0	0		None

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
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)
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?
24e	EEXIT from debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
24f	EEXIT from debug enclave	0	0	D.C.		None
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.TSX(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.TSX(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None

Table 35-53. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
28b	XABORT(Async XAbort, or other)	0	1	1		MODE.TSX(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.TSX(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE.TSX(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE.TSX(InTX=0, TXAbort=1), TraceStop?
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	PIP(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?
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(SIPI LIP); TraceStop?
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(SIPI LIP)
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 35-54, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PwrEvtEn).

Table 35-54. 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	D.C.	D.C.	D.C.		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	D.C.	D.C.	D.C.		See VM exit examples (16[a-z] in Table 35-53) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests CO, or monitor not armed, or VMX virtual-interrupt delivery	D.C.	D.C.	D.C.		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	D.C.	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	D.C.	D.C.	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)	D.C.	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)	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	D.C.	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	D.C.	D.C.	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)	D.C.	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	D.C.	D.C.			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	D.C.	0	0		See "VMX Abort" (cases 16* and 18* in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP
16.10b	VMX abort	D.C.	D.C.	1		See "VMX Abort" (cases 16* and 18* in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	D.C.	0	0		See "RSM to Shutdown" (cases 15[def] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP

Table 35-54. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxE After	Other Dependencies	Packets Output
16.11b	RSM to Shutdown	D.C.	D.C.	1		See "RSM to Shutdown" (cases 15[def] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.12a	INIT (BSP)	D.C.	0	0		See "INIT (BSP)" (cases 30[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP
16.12b	INIT (BSP)	D.C.	D.C.	1		See "INIT (BSP)" (cases 30[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.13a	INIT (AP, goes to Wait-for-SIPI)	D.C.	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP
16.13b	INIT (AP, goes to Wait-for-SIPI)	D.C.	D.C.	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	D.C.	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)	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	D.C.	0	0		See "VM entry" (cases 17[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP

Table 35-54. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxE After	Other Dependencies	Packets Output
16.15b	VM entry to HLT or Shutdown	D.C.	D.C.	1		See “VM entry” (cases 17[a-z] in Table 35-53) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STP-CLK#	D.C.	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#	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	D.C.	D.C.	D.C.		None
16.18	INTR during Cx (x>0)	D.C.	D.C.	D.C.	* 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 35-53) for BranchEn packets that follow.
16.18	SMI during Cx (x>0)	D.C.	D.C.	D.C.	* 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 35-53) for BranchEn packets that follow.
16.19	NMI during Cx (x>0)	D.C.	D.C.	D.C.	* 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 35-53) for BranchEn packets that follow.
16.20	Store to monitored address during Cx (x>0)	D.C.	D.C.	D.C.	* 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)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 35-55, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PTWEn).

Table 35-55. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxE After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	D.C.	D.C.	D.C.		None
16.24b	PTWRITE rm32/64, no fault	D.C.	0	0		None

Table 35-55. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.24d	PTWRITE rm32, no fault	D.C.	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	D.C.	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	D.C.	D.C.	D.C.		See "Exception/fault" (cases 13[a-z] in Table 35-53) 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 Section 35.2.8.3, 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 follow:

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 be calculated as 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:

$$\text{TimeStampValue} = (\text{CoreCrystalClockValue} * P) + \text{AdjustedProcessorCycles} + \text{Software_Offset};$$

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:

$$(\text{CTC}_B - \text{CTC}_A), \text{ where } \text{CTC}_i = \text{MTC}_i[15:8] \ll \text{IA32_RTIT_CTL.MTCFreq} \text{ and } i = A, B.$$

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 Adjust-

edProcessorCycles 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

$CTC_{\text{Delta}}[15:0] = (CTC_{\text{Next}}[15:0] - TMA.CTC[15:0])$, where $CTC_{\text{Next}} = MTC_{\text{Payload}} \ll IA32_RTIT_CTL.MTCFreq$.

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 $P1/CBR_{\text{payload}}$.

Note that stand-alone TSC packets (that is, TSC packets that are not a part of a PSB+) are typically generated only when generation of other timing packets (MTCs and CYCs) has ceased for a period of time. Example scenarios include when Intel PT is re-enabled, or on wake after a sleep state. Thus any calculation of ART or cycle time leading up to a TSC packet will likely result in a discrepancy, which the TSC packet serves to correct.

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 35.5.2.4 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.

33. Updates to Chapter 37, Volume 3D

Change bars show changes to Chapter 37 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

Changes to this chapter: Typo corrections as necessary. Added information to Section 37.9.1.1 "EXITINFO".

CHAPTER 37

ENCLAVE ACCESS CONTROL AND DATA STRUCTURES

37.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT

When an enclave is created, it has a range of linear addresses to which the processor applies enhanced access control. This range is called the ELRANGE (see Section 36.3). When an enclave generates a memory access, the existing IA32 segmentation and paging architecture are applied. Additionally, linear addresses inside the ELRANGE must map to an EPC page otherwise when an enclave attempts to access that linear address a fault is generated.

The EPC pages need not be physically contiguous. System software allocates EPC pages to various enclaves. Enclaves must abide by OS/VMM imposed segmentation and paging policies. OS/VMM-managed page tables and extended page tables provide address translation for the enclave pages. Hardware requires that these pages are properly mapped to EPC (any failure generates an exception).

Enclave entry must happen through specific enclave instructions:

- ENCLU[EENTER], ENCLU[ERESUME].

Enclave exit must happen through specific enclave instructions or events:

- ENCLU[EEXIT], Asynchronous Enclave Exit (AEX).

Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations. As an example a read of an enclave page may result in the return of all one's or return of cyphertext of the cache line. Writing to an enclave page may result in a dropped write or a machine check at a later time. The processor will provide the protections as described in Section 37.4 and Section 37.5 on such accesses.

37.2 TERMINOLOGY

A memory access to the ELRANGE and initiated by an instruction executed by an enclave is called a Direct Enclave Access (Direct EA).

Memory accesses initiated by certain Intel® SGX instruction leaf functions such as ECREATE, EADD, EDBGRD, EDBGWR, ELDU/ELDB, EWB, EREMOVE, EENTER, and ERESUME to EPC pages are called Indirect Enclave Accesses (Indirect EA). Table 37-1 lists additional details of the indirect EA of SGX1 and SGX2 extensions.

Direct EAs and Indirect EAs together are called Enclave Accesses (EAs).

Any memory access that is not an Enclave Access is called a non-enclave access.

37.3 ACCESS-CONTROL REQUIREMENTS

Enclave accesses have the following access-control attributes:

- All memory accesses must conform to segmentation and paging protection mechanisms.
- Code fetches from inside an enclave to a linear address outside that enclave result in a #GP(0) exception.
- Shadow-stack-load or shadow-stack-store from inside an enclave to a linear address outside that enclave results in a #GP(0) exception.
- Non-enclave accesses to EPC memory result in undefined behavior. EPC memory is protected as described in Section 37.4 and Section 37.5 on such accesses.
- EPC pages of page types PT_REG, PT_TCS and PT_TRIM must be mapped to ELRANGE at the linear address specified when the EPC page was allocated to the enclave using ENCLS[EADD] or ENCLS[EAUG] leaf functions. Enclave accesses through other linear address result in a #PF with the PFEC.SGX bit set.

- Direct EAs to any EPC pages must conform to the currently defined security attributes for that EPC page in the EPCM. These attributes may be defined at enclave creation time (EADD) or when the enclave sets them using SGX2 instructions. The failure of these checks results in a #PF with the PFEC.SGX bit set.
 - Target page must belong to the currently executing enclave.
 - Data may be written to an EPC page if the EPCM allow write access.
 - Data may be read from an EPC page if the EPCM allow read access.
 - Instruction fetches from an EPC page are allowed if the EPCM allows execute access.
 - Shadow-stack-load from an EPC page and shadow-stack-store to an EPC page are allowed only if the page type is PT_SS_FIRST or PT_SS_REST.
 - Data writes that are not shadow-stack-store are not allowed if the EPCM page type is PT_SS_FIRST or PT_SS_REST.
 - Target page must not have a restricted page type¹ (PT_SECS, PT_TCS, PT_VA, or PT_TRIM).
 - The EPC page must not be BLOCKED.
 - The EPC page must not be PENDING.
 - The EPC page must not be MODIFIED.

37.4 SEGMENT-BASED ACCESS CONTROL

Intel SGX architecture does not modify the segment checks performed by a logical processor. All memory accesses arising from a logical processor in protected mode (including enclave access) are subject to segmentation checks with the applicable segment register.

To ensure that outside entities do not modify the enclave's logical-to-linear address translation in an unexpected fashion, ENCLU[EENTER] and ENCLU[ERESUME] check that CS, DS, ES, and SS, if usable (i.e., not null), have segment base value of zero. A non-zero segment base value for these registers results in a #GP(0).

On enclave entry either via EENTER or ERESUME, the processor saves the contents of the external FS and GS registers, and loads these registers with values stored in the TCS at build time to enable the enclave's use of these registers for accessing the thread-local storage inside the enclave. On EEXIT and AEX, the contents at time of entry are restored. On AEX, the values of FS and GS are saved in the SSA frame. On ERESUME, FS and GS are restored from the SSA frame. The details of these operations can be found in the descriptions of EENTER, ERESUME, EEXIT, and AEX flows.

37.5 PAGE-BASED ACCESS CONTROL

37.5.1 Access-control for Accesses that Originate from non-SGX Instructions

Intel SGX builds on the processor's paging mechanism to provide page-granular access-control for enclave pages. Enclave pages are designed to be accessible only from inside the currently executing enclave if they belong to that enclave. In addition, enclave accesses must conform to the access control requirements described in Section 37.3. or through certain Intel SGX instructions. Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations.

37.5.2 Memory Accesses that Split across ELRANGE

Memory data accesses are allowed to split across ELRANGE (i.e., a part of the access is inside ELRANGE and a part of the access is outside ELRANGE) while the processor is inside an enclave. If an access splits across ELRANGE, the

1. EPCM may allow write, read or execute access only for pages with page type PT_REG.

processor splits the access into two sub-accesses (one inside ELRANGE and the other outside ELRANGE), and each access is evaluated. A code-fetch access that splits across ELRANGE results in a #GP due to the portion that lies outside of the ELRANGE.

37.5.3 Implicit vs. Explicit Accesses

Memory accesses originating from Intel SGX instruction leaf functions are categorized as either explicit accesses or implicit accesses. Table 37-1 lists the implicit and explicit memory accesses made by Intel SGX leaf functions.

37.5.3.1 Explicit Accesses

Accesses to memory locations provided as explicit operands to Intel SGX instruction leaf functions, or their linked data structures are called explicit accesses.

Explicit accesses are always made using logical addresses. These accesses are subject to segmentation, paging, extended paging, and APIC-virtualization checks, and trigger any faults/exit associated with these checks when the access is made.

The interaction of explicit memory accesses with data breakpoints is leaf-function-specific, and is documented in Section 42.3.4.

37.5.3.2 Implicit Accesses

Accesses to data structures whose physical addresses are cached by the processor are called implicit accesses. These addresses are not passed as operands of the instruction but are implied by use of the instruction.

These accesses do not trigger any access-control faults/exits or data breakpoints. Table 37-1 lists memory objects that Intel SGX instruction leaf functions access either by explicit access or implicit access. The addresses of explicit access objects are passed via register operands with the second through fourth column of Table 37-1 matching implicitly encoded registers RBX, RCX, RDX.

Physical addresses used in different implicit accesses are cached via different instructions and for different durations. The physical address of SECS associated with each EPC page is cached at the time the page is added to the enclave via ENCLS[EADD] or ENCLS[EAUG], or when the page is loaded to EPC via ENCLS[ELDB] or ENCLS[ELDU]. This binding is severed when the corresponding page is removed from the EPC via ENCLS[EREMOVE] or ENCLS[EWB]. Physical addresses of TCS and SSA pages are cached at the time of most-recent enclave entry. Exit from an enclave (ENCLU[EEXIT] or AEX) flushes this caching. Details of Asynchronous Enclave Exit is described in Chapter 39.

The physical addresses that are cached for use by implicit accesses are derived from logical (or linear) addresses after checks such as segmentation, paging, EPT, and APIC virtualization checks. These checks may trigger exceptions or VM exits. Note, however, that such exception or VM exits may not occur after a physical address is cached and used for an implicit access.

Table 37-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions

Instr. Leaf	Enum.	Explicit 1	Explicit 2	Explicit 3	Implicit
EACCEPT	SGX2	SECINFO	EPCPAGE		SECS
EACCEPTCOPY	SGX2	SECINFO	EPCPAGE (Src)	EPCPAGE (Dst)	
EADD	SGX1	PAGEINFO and linked structures	EPCPAGE		
EAUG	SGX2	PAGEINFO and linked structures	EPCPAGE		SECS
EBLOCK	SGX1	EPCPAGE			SECS
ECREATE	SGX1	PAGEINFO and linked structures	EPCPAGE		
EDBGRD	SGX1	EPCADDR	Destination		SECS
EDBGWR	SGX1	EPCADDR	Source		SECS
EDECVIRTCHILD	OVERSUB	EPCPAGE	SECS		
EENTER	SGX1	TCS and linked SSA			SECS

Table 37-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions (Contd.)

Instr. Leaf	Enum.	Explicit 1	Explicit 2	Explicit 3	Implicit
EEXIT	SGX1				SECS, TCS
EEXTEND	SGX1	SECS	EPCPAGE		
EGETKEY	SGX1	KEYREQUEST	KEY		SECS
EINCVIRTCHILD	OVERSUB	EPCPAGE	SECS		
EINIT	SGX1	SIGSTRUCT	SECS	EINITTOKEN	
ELDB/ELDU	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	
ELDBC/ELDUC	OVERSUB	PAGEINFO and linked structures	EPCPAGE	VAPAGE	
EMODPE	SGX2	SECINFO	EPCPAGE		
EMODPR	SGX2	SECINFO	EPCPAGE		SECS
EMODT	SGX2	SECINFO	EPCPAGE		SECS
EPA	SGX1	EPCADDR			
ERDINFO	OVERSUB	RDINFO	EPCPAGE		
EREMOVE	SGX1	EPCPAGE			SECS
EREPORT	SGX1	TARGETINFO	REPORTDATA	OUTPUTDATA	SECS
ERESUME	SGX1	TCS and linked SSA			SECS
ESETCONTEXT	OVERSUB		SECS	ContextValue	
ETRACK	SGX1	EPCPAGE			
ETRACKC	OVERSUB		EPCPAGE		
EWB	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	SECS
Asynchronous Enclave Exit*					SECS, TCS, SSA

*Details of Asynchronous Enclave Exit (AEX) is described in Section 39.4

37.6 INTEL® SGX DATA STRUCTURES OVERVIEW

Enclave operation is managed via a collection of data structures. Many of the top-level data structures contain sub-structures. The top-level data structures relate to parameters that may be used in enclave setup/maintenance, by Intel SGX instructions, or AEX event. The top-level data structures are:

- SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State Save Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)
- Paging Crypto MetaData (PCMD)
- Enclave Signature Structure (SIGSTRUCT)
- EINIT Token Structure (EINITTOKEN)
- Report Structure (REPORT)
- Report Target Info (TARGETINFO)
- Key Request (KEYREQUEST)
- Version Array (VA)
- Enclave Page Cache Map (EPCM)
- Read Info (RDINFO)

Details of the top-level data structures and associated sub-structures are listed in Section 37.7 through Section 37.20.

37.7 SGX ENCLAVE CONTROL STRUCTURE (SECS)

The SECS data structure requires 4K-Bytes alignment.

Table 37-2. Layout of SGX Enclave Control Structure (SECS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
SIZE	0	8	Size of enclave in bytes; must be power of 2.
BASEADDR	8	8	Enclave Base Linear Address must be naturally aligned to size.
SSAFRAMESIZE	16	4	Size of one SSA frame in pages, including XSAVE, pad, GPR, and MISC (if CPUID.(EAX=12H, ECX=0):EBX != 0).
MISCSELECT	20	4	Bit vector specifying which extended features are saved to the MISC region (see Section 37.7.2) of the SSA frame when an AEX occurs.
CET_LEG_BITMAP_OFFSET	24	8	Page aligned offset of legacy code page bitmap from enclave base. Software is expected to program this offset such that the entire bitmap resides in the ELRANGE when legacy compatibility mode for indirect branch tracking is enabled. However this is not enforced by the hardware. This field exists when CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] is enumerated as 1, else it is reserved.
CET_ATTRIBUTES	32	1	CET feature attributes of the enclave; see Table 37-5. This field exists when CPUID.(EAX=12,ECX=1):EAX[6] is enumerated as 1, else it is reserved.
RESERVED	24	24	
ATTRIBUTES	48	16	Attributes of the Enclave, see Table 37-3.
MRENCLAVE	64	32	Measurement Register of enclave build process. See SIGSTRUCT for format.
RESERVED	96	32	
MRSIGNER	128	32	Measurement Register extended with the public key that verified the enclave. See SIGSTRUCT for format.
RESERVED	160	32	
CONFIGID	192	64	Post EINIT configuration identity.
ISVPRODID	256	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
CONFIGSVN	260	2	Post EINIT configuration security version number (SVN).
RESERVED	262	3834	<p>The RESERVED field consists of the following:</p> <ul style="list-style-type: none"> ▪ EID: An 8 byte Enclave Identifier. Its location is implementation specific. ▪ PAD: A 352 bytes padding pattern from the Signature (used for key derivation strings). It's location is implementation specific. ▪ VIRTCHILDCNT: An 8 byte Count of virtual children that have been paged out by a VMM. Its location is implementation specific. ▪ ENCLAVECONTEXT: An 8 byte Enclave context pointer. Its location is implementation specific. ▪ ISVFAMILYID: A 16 byte value assigned to identify the family of products the enclave belongs to. ▪ ISVEXTPRODID: A 16 byte value assigned to identify the product identity of the enclave. ▪ The remaining 3226 bytes are reserved area. <p>The entire 3834 byte field must be cleared prior to executing ECREATE.</p>

37.7.1 ATTRIBUTES

The ATTRIBUTES data structure is comprised of bit-granular fields that are used in the SECS, the REPORT and the KEYREQUEST structures. CPUID.(EAX=12H, ECX=1) enumerates a bitmap of permitted 1-setting of bits in ATTRIBUTES.

Table 37-3. Layout of ATTRIBUTES Structure

Field	Bit Position	Description
INIT	0	This bit indicates if the enclave has been initialized by EINIT. It must be cleared when loaded as part of ECREATE. For EREPORT instruction, TARGET_INFO.ATTRIBUTES[ENIT] must always be 1 to match the state after EINIT has initialized the enclave.
DEBUG	1	If 1, the enclave permit debugger to read and write enclave data using EDBGD and EDBGWR.
MODE64BIT	2	Enclave runs in 64-bit mode.
RESERVED	3	Must be Zero.
PROVISIONKEY	4	Provisioning Key is available from EGETKEY.
EINITTOKEN_KEY	5	EINIT token key is available from EGETKEY.
CET	6	Enable CET attributes. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0 this bit is reserved and must be 0.
KSS	7	Key Separation and Sharing Enabled.
RESERVED	63:8	Must be zero.
XFRM	127:64	XSAVE Feature Request Mask. See Section 41.7.

37.7.2 SECS.MISCSELECT Field

CPUID.(EAX=12H, ECX=0):EBX[31:0] enumerates which extended information that the processor can save into the MISC region of SSA when an AEX occurs. An enclave writer can specify via SIGSTRUCT how to set the SECS.MISCSELECT field. The bit vector of MISCSELECT selects which extended information is to be saved in the MISC region of the SSA frame when an AEX is generated. The bit vector definition of extended information is listed in Table 37-4.

If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, SECS.MISCSELECT field must be all zeros.

The SECS.MISCSELECT field determines the size of MISC region of the SSA frame, see Section 37.9.2.

Table 37-4. Bit Vector Layout of MISCSELECT Field of Extended Information

Field	Bit Position	Description
EXINFO	0	Report information about page fault and general protection exception that occurred inside an enclave.
CPINFO	1	Report information about control protection exception that occurred inside an enclave. When CPUID.(EAX=12H, ECX=0):EBX[1] is 0, this bit is reserved.
Reserved	31:2	Reserved (0).

37.7.3 SECS.CET_ATTRIBUTES Field

The SECS.CET_ATTRIBUTES field can be used by the enclave writer to enable various CET attributes in an enclave. This field exists when CPUID.(EAX=12, ECX=1):EAX[6] is enumerated as 1. Bits 1:0 are defined when CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1, and bits 5:2 are defined when CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1.

Table 37-5. Bit Vector Layout of CET_ATTRIBUTES Field of Extended Information

Field	Bit Position	Description
SH_STK_EN	0	When set to 1, enable shadow stacks.
WR_SHSTK_EN	1	When set to 1, enables the WRSS{D,Q}W instructions.
ENDBR_EN	2	When set to 1, enables indirect branch tracking.
LEG_IW_EN	3	Enable legacy compatibility treatment for indirect branch tracking.
NO_TRACK_EN	4	When set to 1, enables use of no-track prefix for indirect branch tracking.
SUPPRESS_DIS	5	When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.
Reserved	7:6	Reserved (0).

37.8 THREAD CONTROL STRUCTURE (TCS)

Each executing thread in the enclave is associated with a Thread Control Structure. It requires 4K-Bytes alignment.

Table 37-6. Layout of Thread Control Structure (TCS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
STAGE	0	8	Enclave execution state of the thread controlled by this TCS. A value of 0 indicates that this TCS is available for enclave entry. A value of 1 indicates that a processor is currently executing an enclave in the context of this TCS.
FLAGS	8	8	The thread's execution flags (see Section 37.8.1).
OSSA	16	8	Offset of the base of the State Save Area stack, relative to the enclave base. Must be page aligned.
CSSA	24	4	Current slot index of an SSA frame, cleared by EADD and EACCEPT.
NSSA	28	4	Number of available slots for SSA frames.
OENTRY	32	8	Offset in enclave to which control is transferred on EENTER relative to the base of the enclave.
AEP	40	8	The value of the Asynchronous Exit Pointer that was saved at EENTER time.
OFSBASE	48	8	Offset to add to the base address of the enclave for producing the base address of FS segment inside the enclave. Must be page aligned.
OGSBASE	56	8	Offset to add to the base address of the enclave for producing the base address of GS segment inside the enclave. Must be page aligned.
FSLIMIT	64	4	Size to become the new FS limit in 32-bit mode.
GSLIMIT	68	4	Size to become the new GS limit in 32-bit mode.
OCETSSA	72	8	When CPUID.(EAX=12H, ECX=1);EAX[6] is 1, this field provides the offset of the CET state save area from enclave base. When CPUID.(EAX=12H, ECX=1);EAX[6] is 0, this field is reserved and must be 0.
PREVSSP	80	8	When CPUID.(EAX=07H, ECX=00h);ECX[CET_SS] is 1, this field records the SSP at the time of AEX or EEXIT; used to setup SSP on entry. When CPUID.(EAX=07H, ECX=00h);ECX[CET_SS] is 0, this field is reserved and must be 0.
RESERVED	72	4024	Must be zero.

37.8.1 TCS.FLAGS

Table 37-7. Layout of TCS.FLAGS Field

Field	Bit Position	Description
DBGOPTIN	0	If set, allows debugging features (single-stepping, breakpoints, etc.) to be enabled and active while executing in the enclave on this TCS. Hardware clears this bit on EADD. A debugger may later modify it if the enclave's ATTRIBUTES.DEBUG is set.
RESERVED	63:1	

37.8.2 State Save Area Offset (OSSA)

The OSSA points to a stack of State Save Area (SSA) frames (see Section 37.9) used to save the processor state when an interrupt or exception occurs while executing in the enclave.

37.8.3 Current State Save Area Frame (CSSA)

CSSA is the index of the current SSA frame that will be used by the processor to determine where to save the processor state on an interrupt or exception that occurs while executing in the enclave. It is an index into the array of frames addressed by OSSA. CSSA is incremented on an AEX and decremented on an ERESUME.

37.8.4 Number of State Save Area Frames (NSSA)

NSSA specifies the number of SSA frames available for this TCS. There must be at least one available SSA frame when EENTER-ing the enclave or the EENTER will fail.

37.9 STATE SAVE AREA (SSA) FRAME

When an AEX occurs while running in an enclave, the architectural state is saved in the thread's current SSA frame, which is pointed to by TCS.CSSA. An SSA frame must be page aligned, and contains the following regions:

- The XSAVE region starts at the base of the SSA frame, this region contains extended feature register state in an XSAVE/FXSAVE-compatible non-compacted format.
- A Pad region: software may choose to maintain a pad region separating the XSAVE region and the MISC region. Software choose the size of the pad region according to the sizes of the MISC and GPRSGX regions.
- The GPRSGX region. The GPRSGX region is the last region of an SSA frame (see Table 37-8). This is used to hold the processor general purpose registers (RAX ... R15), the RIP, the outside RSP and RBP, RFLAGS and the AEX information.
- The MISC region (If CPUIDEAX=12H, ECX=0):EBX[31:0] != 0). The MISC region is adjacent to the GRPSGX region, and may contain zero or more components of extended information that would be saved when an AEX occurs. If the MISC region is absent, the region between the GPRSGX and XSAVE regions is the pad region that software can use. If the MISC region is present, the region between the MISC and XSAVE regions is the pad region that software can use. See additional details in Section 37.9.2.

Table 37-8. Top-to-Bottom Layout of an SSA Frame

Region	Offset (Byte)	Size (Bytes)	Description
XSAVE	0	Calculate using CPUID leaf ODH information	The size of XSAVE region in SSA is derived from the enclave's support of the collection of processor extended states that would be managed by XSAVE. The enablement of those processor extended state components in conjunction with CPUID leaf ODH information determines the XSAVE region size in SSA.
Pad	End of XSAVE region	Chosen by enclave writer	Ensure the end of GPRSGX region is aligned to the end of a 4KB page.

Table 37-8. Top-to-Bottom Layout of an SSA Frame

Region	Offset (Byte)	Size (Bytes)	Description
MISC	base of GPRSGX - sizeof(MISC)	Calculate from highest set bit of SECS.MISCSELECT	See Section 37.9.2.
GPRSGX	SSAFRAMESIZE - 176	176	See Table 37-9 for layout of the GPRSGX region.

37.9.1 GPRSGX Region

The layout of the GPRSGX region is shown in Table 37-9.

Table 37-9. Layout of GPRSGX Portion of the State Save Area

Field	OFFSET (Bytes)	Size (Bytes)	Description
RAX	0	8	
RCX	8	8	
RDX	16	8	
RBX	24	8	
RSP	32	8	
RBP	40	8	
RSI	48	8	
RDI	56	8	
R8	64	8	
R9	72	8	
R10	80	8	
R11	88	8	
R12	96	8	
R13	104	8	
R14	112	8	
R15	120	8	
RFLAGS	128	8	Flag register.
RIP	136	8	Instruction pointer.
URSP	144	8	Non-Enclave (outside) stack pointer. Saved by EENTER, restored on AEX.
URBP	152	8	Non-Enclave (outside) RBP pointer. Saved by EENTER, restored on AEX.
EXITINFO	160	4	Contains information about exceptions that cause AEXs, which might be needed by enclave software (see Section 37.9.1.1).
RESERVED	164	4	
FSBASE	168	8	FS BASE.
GSBASE	176	8	GS BASE.

37.9.1.1 EXITINFO

EXITINFO contains the information used to report exit reasons to software inside the enclave. It is a 4 byte field laid out as in Table 37-10. The VALID bit is set only for the exceptions conditions which are reported inside an enclave. See Table 37-11 for which exceptions are reported inside the enclave. If the exception condition is not one reported inside the enclave then VECTOR and EXIT_TYPE are cleared.

When a higher priority event, such as SMI, and a pending debug exception occur at the same time when executing inside an enclave, the higher priority event has precedence. As an example for an SMI, the SSA exit info is zero. The debug exception will be delivered upon return from the SMI. In such cases, the EXITINFO field will not contain the information of a debug exception.

Table 37-10. Layout of EXITINFO Field

Field	Bit Position	Description
VECTOR	7:0	Exception number of exceptions reported inside enclave.
EXIT_TYPE	10:8	011b: Hardware exceptions. 110b: Software exceptions. Other values: Reserved.
RESERVED	30:11	Reserved as zero.
VALID	31	0: unsupported exceptions. 1: Supported exceptions. Includes two categories: <ul style="list-style-type: none"> • Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM. • Conditionally supported exception: <ul style="list-style-type: none"> – #PF, #GP if SECS.MISCSELECT.EXINFO = 1. – #CP if SECS.MISCSELECT.CPINFO=1.

37.9.1.2 VECTOR Field Definition

Table 37-11 contains the VECTOR field. This field contains information about some exceptions which occur inside the enclave. These vector values are the same as the values that would be used when vectoring into regular exception handlers. All values not shown are not reported inside an enclave.

Table 37-11. Exception Vectors

Name	Vector #	Description
#DE	0	Divider exception.
#DB	1	Debug exception.
#BP	3	Breakpoint exception.
#BR	5	Bound range exceeded exception.
#UD	6	Invalid opcode exception.
#GP	13	General protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#PF	14	Page fault exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#MF	16	x87 FPU floating-point error.
#AC	17	Alignment check exceptions.
#XM	19	SIMD floating-point exceptions.
#CP	21	Control protection exception. Only reported if SECS.MISCSELECT.CPINFO=1.

37.9.2 MISC Region

The layout of the MISC region is shown in Table 37-12. The number of components that the processor supports in the MISC region corresponds to the bits of CPUID.(EAX=12H, ECX=0):EBX[31:0] set to 1. Each set bit in CPUID.(EAX=12H, ECX=0):EBX[31:0] has a defined size for the corresponding component, as shown in Table 37-12. Enclave writers needs to do the following:

- Decide which MISC region components will be supported for the enclave.
- Allocate an SSA frame large enough to hold the components chosen above.

- Instruct each enclave builder software to set the appropriate bits in SECS.MISCSELECT.

The first component, EXINFO, starts next to the GPRSGX region. Additional components in the MISC region grow in ascending order within the MISC region towards the XSAVE region.

The size of the MISC region is calculated as follows:

- If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISC region is not supported.
- If CPUID.(EAX=12H, ECX=0):EBX[31:0] != 0, the size of MISC region is derived from sum of the highest bit set in SECS.MISCSELECT and the size of the MISC component corresponding to that bit. Offset and size information of currently defined MISC components are listed in Table 37-12. For example, if the highest bit set in SECS.MISCSELECT is bit 0, the MISC region offset is OFFSET(GPRSGX)-16 and size is 16 bytes.
- The processor saves a MISC component *i* in the MISC region if and only if SECS.MISCSELECT[*i*] is 1.

Table 37-12. Layout of MISC region of the State Save Area

MISC Components	OFFSET (Bytes)	Size (Bytes)	Description
EXINFO	Offset(GPRSGX) -16	16	If CPUID.(EAX=12H, ECX=0):EBX[0] = 1, exception information on #GP or #PF that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[0] = 1. If CPUID.(EAX=12H, ECX=0):EBX[1] = 1, exception information on #CP that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[1] = 1.
Future Extension	Below EXINFO	TBD	Reserved. (Zero size if CPUID.(EAX=12H, ECX=0):EBX[31:1] =0).

37.9.2.1 EXINFO Structure

Table 37-13 contains the layout of the EXINFO structure that provides additional information.

Table 37-13. Layout of EXINFO Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MADDR	0	8	If #PF: contains the page fault linear address that caused a page fault. If #GP: the field is cleared. If #CP: the field is cleared.
ERRCD	8	4	Exception error code for either #GP or #PF.
RESERVED	12	4	

37.9.2.2 Page Fault Error Code

Table 37-14 contains page fault error code that may be reported in EXINFO.ERRCD.

Table 37-14. Page Fault Error Code

Name	Bit Position	Description
P	0	Same as non-SGX page fault exception P flag.
W/R	1	Same as non-SGX page fault exception W/R flag.
U/S ¹	2	Always set to 1 (user mode reference).
RSVD	3	Same as non-SGX page fault exception RSVD flag.
I/D	4	Same as non-SGX page fault exception I/D flag.
PK	5	Protection Key induced fault.
RSVD	14:6	Reserved.
SGX	15	EPCM induced fault.
RSVD	31:5	Reserved.

NOTES:

1. Page faults incident to enclave mode that report U/S=0 are not reported in EXINFO.

37.10 CET STATE SAVE AREA FRAME

The CET state save area consists of an array of CET state save frames. The number of CET state save frames is equal to the TCS.NSSA. The current CET SSA frame is indicated by TCS.CSSA. The offset of the CET state save area is specified by TCS.OCETSSA.

Table 37-15. Layout of CET State Save Area Frame

Field	Offset (Bytes)	Size (Bytes)	Description
SSP	0	8	Shadow Stack Pointer. This field is reserved when CPUID.(EAX=7, ECX=0):ECX[CET_SS] is 0.
IB_TRACK_STATE	8	8	Indirect branch tracker state: Bit 0: SUPPRESS - suppressed(1), tracking(0) Bit 1: TRACKER - IDLE (0), WAIT_FOR_ENDBRANCH (1) Bits 63:2 - Reserved This field is reserved when CPUID.(EAX=7, ECX=0):EDX[CET_IBT] is 0.

37.11 PAGE INFORMATION (PAGEINFO)

PAGEINFO is an architectural data structure that is used as a parameter to the EPC-management instructions. It requires 32-Byte alignment.

Table 37-16. Layout of PAGEINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
LINADDR	0	8	Enclave linear address.
SRCPGE	8	8	Effective address of the page where contents are located.
SECINFO/PCMD	16	8	Effective address of the SECINFO or PCMD (for ELDU, ELDB, EWB) structure for the page.
SECS	24	8	Effective address of EPC slot that currently contains the SECS.

37.12 SECURITY INFORMATION (SECINFO)

The SECINFO data structure holds meta-data about an enclave page.

Table 37-17. Layout of SECINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
FLAGS	0	8	Flags describing the state of the enclave page.
RESERVED	8	56	Must be zero.

37.12.1 SECINFO.FLAGS

The SECINFO.FLAGS are a set of fields describing the properties of an enclave page.

Table 37-18. Layout of SECINFO.FLAGS Field

Field	Bit Position	Description
R	0	If 1 indicates that the page can be read from inside the enclave; otherwise the page cannot be read from inside the enclave.
W	1	If 1 indicates that the page can be written from inside the enclave; otherwise the page cannot be written from inside the enclave.
X	2	If 1 indicates that the page can be executed from inside the enclave; otherwise the page cannot be executed from inside the enclave.
PENDING	3	If 1 indicates that the page is in the PENDING state; otherwise the page is not in the PENDING state.
MODIFIED	4	If 1 indicates that the page is in the MODIFIED state; otherwise the page is not in the MODIFIED state.
PR	5	If 1 indicates that a permission restriction operation on the page is in progress, otherwise a permission restriction operation is not in progress.
RESERVED	7:6	Must be zero.
PAGE_TYPE	15:8	The type of page that the SECINFO is associated with.
RESERVED	63:16	Must be zero.

37.12.2 PAGE_TYPE Field Definition

The SECINFO flags and EPC flags contain bits indicating the type of page.

Table 37-19. Supported PAGE_TYPE

TYPE	Value	Description
PT_SECS	0	Page is an SECS.
PT_TCS	1	Page is a TCS.
PT_REG	2	Page is a regular page.
PT_VA	3	Page is a Version Array.
PT_TRIM	4	Page is in trimmed state.
PT_SS_FIRST	5	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.
PT_SS_REST	6	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is not first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.
	All others	Reserved.

37.13 PAGING CRYPTO METADATA (PCMD)

The PCMD structure is used to keep track of crypto meta-data associated with a paged-out page. Combined with PAGEINFO, it provides enough information for the processor to verify, decrypt, and reload a paged-out EPC page. The size of the PCMD structure (128 bytes) is architectural.

EWB calculates the Message Authentication Code (MAC) value and writes out the PCMD. ELDB/U reads the fields and checks the MAC.

The format of PCMD is as follows:

Table 37-20. Layout of PCMD Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
SECINFO	0	64	Flags describing the state of the enclave page; R/W by software.
ENCLAVEID	64	8	Enclave Identifier used to establish a cryptographic binding between paged-out page and the enclave.
RESERVED	72	40	Must be zero.
MAC	112	16	Message Authentication Code for the page, page meta-data and reserved field.

37.14 ENCLAVE SIGNATURE STRUCTURE (SIGSTRUCT)

SIGSTRUCT is a structure created and signed by the enclave developer that contains information about the enclave. SIGSTRUCT is processed by the EINIT leaf function to verify that the enclave was properly built.

SIGSTRUCT includes ENCLAVEHASH as SHA256 digest, as defined in FIPS PUB 180-4. The digests are byte strings of length 32. Each of the 8 HASH dwords is stored in little-endian order.

SIGSTRUCT includes four 3072-bit integers (MODULUS, SIGNATURE, Q1, Q2). Each such integer is represented as a byte strings of length 384, with the most significant byte at the position "offset + 383", and the least significant byte at position "offset".

The (3072-bit integer) SIGNATURE should be an RSA signature, where: a) the RSA modulus (MODULUS) is a 3072-bit integer; b) the public exponent is set to 3; c) the signing procedure uses the EMSA-PKCS1-v1.5 format with DER encoding of the "DigestInfo" value as specified in of PKCS#1 v2.1/RFC 3447.

The 3072-bit integers Q1 and Q2 are defined by:

$$q1 = \text{floor}(\text{Signature}^2 / \text{Modulus});$$

$$q2 = \text{floor}((\text{Signature}^3 - q1 * \text{Signature} * \text{Modulus}) / \text{Modulus});$$

SIGSTRUCT must be page aligned

In column 5 of Table 37-21, 'Y' indicates that this field should be included in the signature generated by the developer.

Table 37-21. Layout of Enclave Signature Structure (SIGSTRUCT)

Field	OFFSET (Bytes)	Size (Bytes)	Description	Signed
HEADER	0	16	Must be byte stream 06000000E100000000001000000000H	Y
VENDOR	16	4	Intel Enclave: 00008086H Non-Intel Enclave: 00000000H	Y
DATE	20	4	Build date is yyyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31	Y
HEADER2	24	16	Must be byte stream 010100006000000006000000001000000H	Y
SWDEFINED	40	4	Available for software use.	Y
RESERVED	44	84	Must be zero.	Y
MODULUS	128	384	Module Public Key (keylength=3072 bits).	N
EXPONENT	512	4	RSA Exponent = 3.	N
SIGNATURE	516	384	Signature over Header and Body.	N
MISCSELECT*	900	4	Bit vector specifying Extended SSA frame feature set to be used.	Y
MISCMASK*	904	4	Bit vector mask of MISCSELECT to enforce.	Y

Table 37-21. Layout of Enclave Signature Structure (SIGSTRUCT)

Field	OFFSET (Bytes)	Size (Bytes)	Description	Signed
CET_ATTRIBUTES	908	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Enclave CET attributes that must be set. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.	Y
CET_ATTRIBUTES_MASK	909	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Mask of CET attributes to enforce. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.	Y
RESERVED	910	2	Must be zero.	Y
ISVFAMILYID	912	16	ISV assigned Product Family ID.	Y
ATTRIBUTES	928	16	Enclave Attributes that must be set.	Y
ATTRIBUTEMASK	944	16	Mask of Attributes to enforce.	Y
ENCLAVEHASH	960	32	MRENCLAVE of enclave this structure applies to.	Y
RESERVED	992	16	Must be zero.	Y
ISVEXTPRODID	1008	16	ISV assigned extended Product ID.	Y
ISVPRODID	1024	2	ISV assigned Product ID.	Y
ISVSVN	1026	2	ISV assigned SVN (security version number).	Y
RESERVED	1028	12	Must be zero.	N
Q1	1040	384	Q1 value for RSA Signature Verification.	N
Q2	1424	384	Q2 value for RSA Signature Verification.	N
<p>* If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISCSELECT must be 0. If CPUID.(EAX=12H, ECX=0):EBX[31:0] !=0, enclave writers must specify MISCSELECT such that each cleared bit in MISCMASK must also specify the corresponding bit as 0 in MISCSELECT.</p>				

37.15 EINIT TOKEN STRUCTURE (EINITTOKEN)

The EINIT token is used by EINIT to verify that the enclave is permitted to launch. EINIT token is generated by an enclave in possession of the EINITTOKEN key (the Launch Enclave).

EINIT token must be 512-Byte aligned.

Table 37-22. Layout of EINIT Token (EINITTOKEN)

Field	OFFSET (Bytes)	Size (Bytes)	MACed	Description
Valid	0	4	Y	Bit 0: 1: Valid; 0: Invalid. All other bits reserved.
RESERVED	4	44	Y	Must be zero.
ATTRIBUTES	48	16	Y	ATTRIBUTES of the Enclave.
MRENCLAVE	64	32	Y	MRENCLAVE of the Enclave.
RESERVED	96	32	Y	Reserved.
MRSIGNER	128	32	Y	MRSIGNER of the Enclave.
RESERVED	160	32	Y	Reserved.
CPUSVNLE	192	16	N	Launch Enclave's CPUSVN.
ISVPRODIDLE	208	02	N	Launch Enclave's ISVPRODID.
ISVSVNLE	210	02	N	Launch Enclave's ISVSVN.
CET_MASKED_ATTRIBUTES_LE	212	1	N	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Launch enclaves masked CET attributes. This should be set to LE's CET_ATTRIBUTES masked with CET_ATTRIBUTES_MASK of the LE's KEYREQUEST. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved.
RESERVED	213	23	N	Reserved.
MASKEDMISCSELECTLE	236	4		Launch Enclave's MASKEDMISCSELECT: set by the LE to the resolved MISCSELECT value, used by EGETKEY (after applying KEYREQUEST's masking).
MASKEDATTRIBUTESLE	240	16	N	Launch Enclave's MASKEDATTRIBUTES: This should be set to the LE's ATTRIBUTES masked with ATTRIBUTEMASK of the LE's KEYREQUEST.
KEYID	256	32	N	Value for key wear-out protection.
MAC	288	16	N	Message Authentication Code on EINITTOKEN using EINITTOKEN_KEY.

37.16 REPORT (REPORT)

The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

Table 37-23. Layout of REPORT

Field	OFFSET (Bytes)	Size (Bytes)	Description
CPUSVN	0	16	The security version number of the processor.
MISCSELECT	16	4	Bit vector specifying which extended features are saved to the MISC region of the SSA frame when an AEX occurs.
CET_ATTRIBUTES	20	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field reports the CET_ATTRIBUTES of the Enclave. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.
RESERVED	20	12	Zero.
ISVEXTNPRODID	32	16	The value of SECS.ISVEXTPRODID.
ATTRIBUTES	48	16	ATTRIBUTES of the Enclave. See Section 37.7.1.
MRENCLAVE	64	32	The value of SECS.MRENCLAVE.
RESERVED	96	32	Zero.
MRSIGNER	128	32	The value of SECS.MRSIGNER.
RESERVED	160	32	Zero.

Table 37-23. Layout of REPORT

Field	OFFSET (Bytes)	Size (Bytes)	Description
CONFIGID	192	64	Value provided by SW to identify enclave's post EINIT configuration.
ISVPRODID	256	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
CONFIGSVN	260	2	Value provided by SW to indicate expected SVN of enclave's post EINIT configuration.
RESERVED	262	42	Zero.
ISVFAMILYID	304	16	The value of SECS.ISVFAMILYID.
REPORTDATA	320	64	Data provided by the user and protected by the REPORT's MAC, see Section 37.16.1.
KEYID	384	32	Value for key wear-out protection.
MAC	416	16	Message Authentication Code on the report using report key.

37.16.1 REPORTDATA

REPORTDATA is a 64-Byte data structure that is provided by the enclave and included in the REPORT. It can be used to securely pass information from the enclave to the target enclave.

37.17 REPORT TARGET INFO (TARGETINFO)

This structure is an input parameter to the EREPORT leaf function. The address of TARGETINFO is specified as an effective address in RBX. It is used to identify the target enclave which will be able to cryptographically verify the REPORT structure returned by EREPORT. TARGETINFO must be 512-Byte aligned.

Table 37-24. Layout of TARGETINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MEASUREMENT	0	32	The MRENCLAVE of the target enclave.
ATTRIBUTES	32	16	The ATTRIBUTES field of the target enclave.
CET_ATTRIBUTES	48	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the CET_ATTRIBUTES field of the target enclave. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved.
RESERVED	49	1	Must be zero.
CONFIGSVN	50	2	CONFIGSVN of the target enclave.
MISCSELECT	52	4	The MISCSELECT of the target enclave.
RESERVED	56	8	Must be zero.
CONFIGID	64	64	CONFIGID of target enclave.
RESERVED	128	384	Must be zero.

37.18 KEY REQUEST (KEYREQUEST)

This structure is an input parameter to the EGETKEY leaf function. It is passed in as an effective address in RBX and must be 512-Byte aligned. It is used for selecting the appropriate key and any additional parameters required in the derivation of that key.

Table 37-25. Layout of KEYREQUEST Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
KEYNAME	0	2	Identifies the Key Required.
KEYPOLICY	2	2	Identifies which inputs are required to be used in the key derivation.
ISVSVN	4	2	The ISV security version number that will be used in the key derivation.
CET_ATTRIBUTES_MASK	6	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides a mask that defines which CET_ATTRIBUTES bits will be included in key derivation. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, then this field is reserved and must be 0.
RESERVED	7	1	Must be zero.
CPUSVN	8	16	The security version number of the processor used in the key derivation.
ATTRIBUTEMASK	24	16	A mask defining which ATTRIBUTES bits will be included in key derivation.
KEYID	40	32	Value for key wear-out protection.
MISCMASK	72	4	A mask defining which MISCSELECT bits will be included in key derivation.
CONFIGSVN	76	2	Identifies which enclave Configuration's Security Version should be used in key derivation.
RESERVED	78	434	

37.18.1 KEY REQUEST KeyNames

Table 37-26. Supported KEYName Values

Key Name	Value	Description
EINIT_TOKEN_KEY	0	EINIT_TOKEN key
PROVISION_KEY	1	Provisioning Key
PROVISION_SEAL_KEY	2	Provisioning Seal Key
REPORT_KEY	3	Report Key
SEAL_KEY	4	Seal Key
	All others	Reserved

37.18.2 Key Request Policy Structure

Table 37-27. Layout of KEYPOLICY Field

Field	Bit Position	Description
MRENCLAVE	0	If 1, derive key using the enclave's MRENCLAVE measurement register.
MRSIGNER	1	If 1, derive key using the enclave's MRSIGNER measurement register.
NOISVPRODID	2	If 1, derive key WITHOUT using the enclave's ISVPRODID value.
CONFIGID	3	If 1, derive key using the enclave's CONFIGID value.
ISVFAMILYID	4	If 1, derive key using the enclave ISVFAMILYID value.
ISVEXTPRODID	5	If 1, derive key using enclave's ISVEXTPRODID value.
RESERVED	15:6	Must be zero.

37.19 VERSION ARRAY (VA)

In order to securely store the versions of evicted EPC pages, Intel SGX defines a special EPC page type called a Version Array (VA). Each VA page contains 512 slots, each of which can contain an 8-byte version number for a page evicted from the EPC. When an EPC page is evicted, software chooses an empty slot in a VA page; this slot receives the unique version number of the page being evicted. When the EPC page is reloaded, there must be a VA slot that must hold the version of the page. If the page is successfully reloaded, the version in the VA slot is cleared.

VA pages can be evicted, just like any other EPC page. When evicting a VA page, a version slot in some other VA page must be used to hold the version for the VA being evicted. A Version Array Page must be 4K-Bytes aligned.

Table 37-28. Layout of Version Array Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
Slot 0	0	8	Version Slot 0
Slot 1	8	8	Version Slot 1
...			
Slot 511	4088	8	Version Slot 511

37.20 ENCLAVE PAGE CACHE MAP (EPCM)

EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds exactly one entry for each page that is currently loaded into the EPC. EPCM is not accessible by software, and the layout of EPCM fields is implementation specific.

Table 37-29. Content of an Enclave Page Cache Map Entry

Field	Description
VALID	Indicates whether the EPCM entry is valid.
R	Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.
W	Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.
X	Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.
PT	EPCM page type (PT_SECS, PT_TCS, PT_REG, PT_VA, PT_TRIM, PT_SS_FIRST, PT_SS_REST).
ENCLAVESECS	SECS identifier of the enclave to which the EPC page belongs.
ENCLAVEADDRESS	Linear enclave address of the EPC page.
BLOCKED	Indicates whether the EPC page is in the blocked state.
PENDING	Indicates whether the EPC page is in the pending state.
MODIFIED	Indicates whether the EPC page is in the modified state.
PR	Indicates whether the EPC page is in a permission restriction state.

37.21 READ INFO (RDINFO)

The RDINFO structure contains status information about an EPC page. It must be aligned to 32-Bytes.

Table 37-30. Layout of RDINFO Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
STATUS	0	8	Page status information.
FLAGS	8	8	EPCM state of the page.
ENCLAVECONTEXT	16	8	Context pointer describing the page's parent location.

37.21.1 RDINFO Status Structure

Table 37-31. Layout of RDINFO STATUS Structure

Field	Bit Position	Description
CHILDPRESENT	0	Indicates that the page has one or more child pages present (always zero for non-SECS pages). In VMX non-root operation includes the presence of virtual children.
VIRTCHLDPRESENT	1	Indicates that the page has one or more virtual child pages present (always zero for non-SECS pages). In VMX non-root operation this value is always zero.
RESERVED	63:2	

37.21.2 RDINFO Flags Structure

Table 37-32. Layout of RDINFO FLAGS Structure

Field	Bit Position	Description
R	0	Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.
W	1	Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.
X	2	Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.
PENDING	3	Indicates whether the EPC page is in the pending state.
MODIFIED	4	Indicates whether the EPC page is in the modified state.
PR	5	Indicates whether the EPC page is in a permission restriction state.
RESERVED	7:6	
PAGE_TYPE	15:8	Indicates the page type of the EPC page.
RESERVED	62:16	
BLOCKED	63	Indicates whether the EPC page is in the blocked state.

34. 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: Typo corrections as necessary.

CHAPTER 40 SGX INSTRUCTION REFERENCES

This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor), ENCLU (user), and the ENCLV (virtualization operation) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

40.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION

ENCLS, ENCLU and ENCLV instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

40.1.1 ENCLS Register Usage Summary

Table 40-1 summarizes the implicit register usage of supervisor mode enclave instructions.

Table 40-1. Register Usage of Privileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
ECREATE	00H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EADD	01H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EINIT	02H (In)	SIGSTRUCT (In, EA)	SECS (In, EA)	EINITTOKEN (In, EA)
EREMOVE	03H (In)		EPCPAGE (In, EA)	
EDBGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
ERDINFO	010H (In)	RDINFO (In, EA*)	EPCPAGE (In, EA)	
ETRACKC	011H (In)		EPCPAGE (In, EA)	
ELDBC	012H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDUC	013H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)

EA: Effective Address

40.1.2 ENCLU Register Usage Summary

Table 40-2 summarizes the implicit register usage of user mode enclave instructions.

Table 40-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDY
EReport	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGetKey	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EEnter	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
EResume	03H (In)	TCS (In, EA)	AEP (In, EA)	
EExit	04H (In)	Target (In, EA)	Current AEP (Out)	
EAccept	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EModPE	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EAcceptCopy	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EA: Effective Address				

40.1.3 ENCLV Register Usage Summary

Table 40-3 summarizes the implicit register usage of virtualization operation enclave instructions.

Table 40-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDY
EDECvirtChild	00H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
EINCvirtChild	01H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
ESETCONTEXT	02H (In)		EPCPAGE (In, EA)	Context Value (In, EA)
EA: Effective Address				

40.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 40-4 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

Table 40-4. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_BLKSTATE	3	EBLOCK
SGX_INVALID_MEASUREMENT	4	EINIT
SGX_NOTBLOCKABLE	5	EBLOCK
SGX_PG_INVLD	6	EBLOCK, ERDINFO, ETRACKC
SGX_EPC_PAGE_CONFLICT	7	EBLOCK, EMODPR, EMODT, ERDINFO, EDECvirtChild, EINCvirtChild, ELDBC, ELDUC, ESETCONTEXT, ETRACKC

Table 40-4. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
SGX_INVALID_SIGNATURE	8	EINIT
SGX_MAC_COMPARE_FAIL	9	ELDB, ELDU, ELDBC, ELDUC
SGX_PAGE_NOT_BLOCKED	10	EWB
SGX_NOT_TRACKED	11	EWB, EACCEPT
SGX_VA_SLOT_OCCUPIED	12	EWB
SGX_CHILD_PRESENT	13	EWB, EREMOVE
SGX_ENCLAVE_ACT	14	EREMOVE
SGX_ENTRYEPOCH_LOCKED	15	EBLOCK
SGX_INVALID_EINITTOKEN	16	EINIT
SGX_PREV_TRK_INCMPL	17	ETRACK, ETRACKC
SGX_PG_IS_SECS	18	EBLOCK
SGX_PAGE_ATTRIBUTES_MISMATCH	19	EACCEPT, EACCEPTCOPY
SGX_PAGE_NOT_MODIFIABLE	20	EMODPR, EMODT
SGX_PAGE_NOT_DEBUGGABLE	21	EDBGRD, EDBGWR
SGX_INVALID_COUNTER	25	EDECVIRTCHILD
SGX_PG_NONEPC	26	ERDINFO
SGX_TRACK_NOT_REQUIRED	27	ETRACKC
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

40.1.5 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 40-5 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

Table 40-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_EL RANGE	128	LP
CR_SAVE_TF	1	LP
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP

Table 40-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNEREPOCH	128	PACKAGE
CR_SAVE_XCRO	64	LP
CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE
CR_CET_SAVE_AREA_PA	64	LP
CR_ENCLAVE_SS_TOKEN_PA	64	LP
CR_SAVE_IA32_U_CET	64	LP
CR_SAVE_SSP	64	LP

40.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leaves to concurrently operate on the same EPC page.
 - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
 - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX_EPC_PAGE_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

40.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.
- **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.

- **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EXTEND and EINIT. EADD and EEXTEND can't execute concurrently on the same SECS page.

The tables also detail the leaf function's behavior when a conflict happens, i.e., a concurrency requirement is not met. In this case, the leaf function may return an SGX_EPC_PAGE_CONFLICT error code in RAX, or it may cause an exception. In addition, the tables detail those conflicts where a VM Exit may be triggered, and list the Exit Qualification code that is provided in such cases.

Table 40-6. Base Concurrency Restrictions

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target	[DS:RCX]	Shared	#GP	
	SECINFO	[DS:RBX]	Concurrent		
EACCEPTCOPY	Target	[DS:RCX]	Concurrent		
	Source	[DS:RDX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EADD	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EAUG	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EBLOCK	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
ECREATE	SECS	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EDBGGRD	Target	[DS:RCX]	Shared	#GP	
EDBGWR	Target	[DS:RCX]	Shared	#GP	
EDECVRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EENTERTCS	SECS	[DS:RBX]	Shared	#GP	
EEXIT			Concurrent		
EEXTEND	Target	[DS:RCX]	Shared	#GP	
	SECS	[DS:RBX]	Concurrent		
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		
	OUTPUTDATA	[DS:RCX]	Concurrent		
EINCVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EINIT	SECS	[DS:RCX]	Shared	#GP	
ELDB/ELDU	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	

Table 40-6. Base Concurrency Restrictions

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDLBC/ELDUC	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA	[DS:RDX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	SGX_EPC_PAGE_CONFLICT	
EMODPE	Target	[DS:RCX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EMODPR	Target	[DS:RCX]	Shared	#GP	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
EPA	VA	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
ERDINFO	Target	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
EREMOVE	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		
	REPORTDATA	[DS:RCX]	Concurrent		
	OUTPUTDATA	[DS:RDX]	Concurrent		
ERESUME	TCS	[DS:RBX]	Shared	#GP	
ESETCONTEXT	SECS	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
ETRACK	SECS	[DS:RCX]	Shared	#GP	
ETRACKC	Target	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS	Implicit	Concurrent		
EWB	Source	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	

Table 40-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EACCEPTCOPY	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	

Table 40-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Exclusive	#GP	Concurrent	
EAUG	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EBLOCK	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ECREATE	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGDR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGWR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDECVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EENTERTCS	SECS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EEXIT			Concurrent		Concurrent		Concurrent	
EEXTEND	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINCVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINIT	SECS	[DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	
ELDB/ELDU	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EDLBC/ELDUC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EMODPE	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EMODPR	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EPA	VA	[DS:RCX]	Concurrent		Concurrent		Concurrent	

Table 40-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREMOVE	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
ERESUME	TCS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
ESETCONTEXT	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ETRACK	SECS	[DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT ¹
ETRACKC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	Implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT ¹
EWB	Source	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	

NOTES:

1. SGX_CONFLICT VM Exit Qualification =TRACKING_RESOURCE_CONFLICT.

40.2 INTEL® SGX INSTRUCTION REFERENCE

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	Z0	V/V	NA	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
Z0	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 an 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/66H/REP/VEX prefixes are 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/66H/REP/VEX prefixes are 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	Z0	V/V	NA	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
Z0	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, as is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE := 0;
```

```
IF TSX_ACTIVE
```

```
    THEN GOTO TSX_ABORT_PROCESSING; FI;
```

(* If enclosing app has CET indirect branch tracking enabled then if it is not ERESUME leaf cause a #CP fault *)

(* If the ERESUME is not successful it will leave tracker in WAIT_FOR_ENDBRANCH *)

```
TRACKER = (CPL == 3) ? IA32_U_CET.TRACKER : IA32_S_CET.TRACKER
```

```
IF EndbranchEnabledAndNotSuppressed(CPL) and TRACKER = WAIT_FOR_ENDBRANCH and  
(EAX != ERESUME or CR0.TS or (in SMM) or (CPUID.SGX_LEAF.0:EAX.SE1 = 0) or (CPL < 3))
```

```
    THEN
```

```
        Handle CET State machine violation          (* see Section 18.3.6, "Legacy Compatibility Treatment" in the  
                                                    Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. *)
```

```
    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/66H/REP/VEX prefixes are 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/66H/REP/VEX prefixes are 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.

ENCLV—Execute an Enclave VMM Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 01 C0 ENCLV	Z0	V/V	NA	This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
Z0	NA	NA	NA	See Section 40.3

Description

The ENCLV instruction invokes the virtualization SGX leaf functions for managing enclaves in a virtualized environment. 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 non 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLV instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, if it is executed in system-management mode (SMM), or not in VMX operation. 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.

Software in VMX root mode of operation can enable execution of the ENCLV instruction in VMX non-root mode by setting enable ENCLV execution control in the VMCS. If enable ENCLV execution control in the VMCS is clear, execution of the ENCLV instruction in VMX non-root mode results in #UD.

When execution of ENCLV instruction in VMX non-root mode is enabled, software in VMX root operation can intercept the invocation of various ENCLV leaf functions in VMX non-root operation by setting the corresponding bits in the ENCLV-exiting bitmap.

Addresses and operands are 32 bits in 32-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.

Segment override prefixes and address-size override prefixes are ignored, as 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.OSS = 0

THEN #UD; FI;

IF not in VMX Operation or (IA32_EFER.LMA = 1 and CS.L = 0)

THEN #UD; FI;

IF (CPL > 0)

THEN #UD; FI;

IF in VMX non-root operation

IF “enable ENCLV exiting” VM-execution control is 1

THEN

IF EAX < 63 and ENCLV_exiting_bitmap[EAX] = 1 or EAX > 62 and ENCLV_exiting_bitmap[63] = 1

THEN VM exit;

FI;

ELSE

#UD; FI;

FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
THEN #GP(0); FI;

IF (EAX is an 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/66H/REP/VEX prefixes are used. If current privilege level is not 0. If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 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	ENCLV is not recognized in real mode.
-----	---------------------------------------

Virtual-8086 Mode Exceptions

#UD	ENCLV 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/66H/REP/VEX prefixes are used. If current privilege level is not 0. If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 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.

40.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EADD—Add a Page to an Uninitialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLS[EADD]	IR	V/V	SGX1	This leaf function adds a page to an uninitialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

Description

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

EADD Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EADD Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
The SECS has been initialized.	The specified enclave offset is outside of the enclave address space.

Concurrency Restrictions

Table 40-8. Base Concurrency Restrictions of EADD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EADD	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Table 40-9. Additional Concurrency Restrictions of EADD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EADD Operational Flow

Name	Type	Size (bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

IF (DS:RBX is not 32Byte Aligned)
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_SECINFO := DS:RBX.SECINFO;
TMP_LINADDR := DS:RBX.LINADDR;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECS is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned or TMP_LINADDR is not 4KByte aligned)
THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
THEN #PF(DS:TMP_SECS); FI;

SCRATCH_SECINFO := DS:TMP_SECINFO;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero or

```

!(SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
THEN #GP(0); FI;

```

```

(* If PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST OR
SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST) AND CR4.CET == 0)
THEN #GP(0); FI;

```

```

(* Check the EPC page for concurrency *)
IF (EPC page is not available for EADD)
THEN
    IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
        THEN
            VMCS.Exit_reason := SGX_CONFLICT;
            VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
            VMCS.Exit_qualification.error := 0;
            VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
            VMCS.Guest-linear_address := DS:RCX;
            Deliver VMEXIT;
        ELSE
            #GP(0);
    FI;
FI;

```

```

IF (EPCM(DS:RCX).VALID ≠ 0)
THEN #PF(DS:RCX); FI;

```

```

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
THEN #GP(0); FI;

```

```

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
THEN #PF(DS:TMP_SECS); FI;

```

```

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];

```

```

CASE (SCRATCH_SECINFO.FLAGS.PT)

```

```

PT_TCS:
    IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
    IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
        ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
    (* Ensure TCS.PREVSSP is zero *)
    IF (CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1) and (DS:RCX.PREVSSP != 0) #GP(0); FI;
    BREAK;

```

```

PT_REG:
    IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); FI;
    BREAK;

```

```

PT_SS_FIRST:

```

```

PT_SS_REST:

```

```

(* SS pages cannot be created on first or last page of ELRANGE *)

```

```

IF ( TMP_LINADDR = DS:TMP_SECS.BASEADDR or TMP_LINADDR = (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
  THEN #GP(0); FI;
IF ( DS:RCX[4087:0] != 0 ) #GP(0); FI;
IF (SCRATCH_SECINFO.FLAGS.PT == PT_SS_FIRST)
  THEN
    (* Check that valid RSTORSSP token exists *)
    IF ( DS:RCX[4095:4088] != ((TMP_LINADDR + 0x1000) | DS:TMP_SECS.ATTRIBUTES.MODE64BIT) ) #GP(0); FI;
    (* Check the 8 bytes are zero *)
    IF ( DS:RCX[4095:4088] != 0 ) #GP(0); FI;
  FI;
IF (SCRATCH_SECINFO.FLAGS.W = 0 OR SCRATCH_SECINFO.FLAGS.R = 0 OR
  SCRATCH_SECINFO.FLAGS.X = 1) #GP(0); FI;
  BREAK;
ESAC;

```

```

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
  THEN #GP(0); FI;

```

```

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
  THEN #GP(0); FI;

```

```

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
  THEN #GP(0); FI;

```

```

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
  THEN
    SCRATCH_SECINFO.FLAGS.R := 0;
    SCRATCH_SECINFO.FLAGS.W := 0;
    SCRATCH_SECINFO.FLAGS.X := 0;
    (DS:RCX).FLAGS.DBGOPTIN := 0; // force TCS.FLAGS.DBGOPTIN off
    DS:RCX.CSSA := 0;
    DS:RCX.AEP := 0;
    DS:RCX.STATE := 0;
  FI;

```

```

(* Add enclave offset and security attributes to MRENCLAVE *)
TMP_ENCLAVEOFFSET := TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] := 0000000044444145H; // "EADD"
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

```

```

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;

```

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
 Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM entry fields *)
 EPCM(DS:RCX).BLOCKED := 0;
 EPCM(DS:RCX).PENDING := 0;
 EPCM(DS:RCX).MODIFIED := 0;
 EPCM(DS:RCX).VALID := 1;

Flags Affected

None

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the DS segment limit.
 If a memory operand is not properly aligned.
 If an enclave memory operand is outside of the EPC.
 If an enclave memory operand is the wrong type.
 If a memory operand is locked.
 If the enclave is initialized.
 If the enclave's MRENCLAVE is locked.
 If the TCS page reserved bits are set.
 If the TCS page PREVSSP field is not zero.
 If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
 If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.

#PF(error code) If a page fault occurs in accessing memory operands.
 If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0) If a memory operand is non-canonical form.
 If a memory operand is not properly aligned.
 If an enclave memory operand is outside of the EPC.
 If an enclave memory operand is the wrong type.
 If a memory operand is locked.
 If the enclave is initialized.
 If the enclave's MRENCLAVE is locked.
 If the TCS page reserved bits are set.
 If the TCS page PREVSSP field is not zero.
 If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
 If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.

#PF(error code) If a page fault occurs in accessing memory operands.
 If the EPC page is valid.

EAUG—Add a Page to an Initialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0DH ENCLS[EAUG]	IR	V/V	SGX2	This leaf function adds a page to an initialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EAUG (In)	Address of a SECFINFO (In)	Address of the destination EPC page (In)

Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPT-COPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EAUG Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

Concurrency Restrictions

Table 40-10. Base Concurrency Restrictions of EAUG

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EAUG	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Table 40-11. Additional Concurrency Restrictions of EAUG

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EAUG	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Concurrent		Concurrent	

Operation**Temp Variables in EAUG Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

```
IF (DS:RBX is not 32Byte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
  THEN #PF(DS:RCX); FI;
```

```
TMP_SECS := DS:RBX.SECS;
TMP_SECINFO := DS:RBX.SECINFO;
IF (DS:RBX.SECINFO is not 0)
  THEN
    IF (DS:TMP_SECINFO is not 64B aligned)
      THEN #GP(0); FI;
```

```
FI;
```

```
TMP_LINADDR := DS:RBX.LINADDR;
```

```
IF ( DS:TMP_SECS is not 4KByte aligned or TMP_LINADDR is not 4KByte aligned )
  THEN #GP(0); FI;
```

```
IF DS:RBX.SRCPAGE is not 0
  THEN #GP(0); FI;
```

```
IF (DS:TMP_SECS does not resolve within an EPC)
  THEN #PF(DS:TMP_SECS); FI;
```

```
(* Check the EPC page for concurrency *)
```



```

IF (EPC page in use)
  THEN
    IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
      THEN
        VMCS.Exit_reason := SGX_CONFLICT;
        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
        VMCS.Guest-linear_address := DS:RCX;
        Deliver VMEXIT;
      ELSE
        #GP(0);
    FI;
  FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
  THEN #PF(DS:RCX); FI;

(* copy SECINFO contents into a scratch SECINFO *)
IF (DS:RBX.SECINFO is 0)
  THEN
    (* allocate and initialize a new scratch secinfo structure *)
    SCRATCH_SECINFO.PT := PT_REG;
    SCRATCH_SECINFO.R := 1;
    SCRATCH_SECINFO.W := 1;
    SCRATCH_SECINFO.X := 0;
    << zero out remaining fields of SCRATCH_SECINFO >>
  ELSE
    (* copy SECINFO contents into scratch secinfo *)
    SCRATCH_SECINFO := DS:TMP_SECINFO;
    (* check SECINFO flags for misconfiguration *)
    (* reserved flags must be zero *)
    (* SECINFO.FLAGS.PT must either be PT_SS_FIRST, or PT_SS_REST *)
    IF ( (SCRATCH_SECINFO reserved fields are not 0) OR
        (SCRATCH_SECINFO.PT is not PT_SS_FIRST, or PT_SS_REST) OR
        ( (SCRATCH_SECINFO.FLAGS.R is 0) OR (SCRATCH_SECINFO.FLAGS.W is 0) OR (SCRATCH_SECINFO.FLAGS.X is 1) ) )
      THEN #GP(0); FI;
  FI;

(* Check if PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) AND CR4.CET == 0 )
  THEN #GP(0); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
  THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
  THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
  THEN #GP(0); FI;

(* Check the enclave offset is within the enclave linear address space *)

```

```
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
  THEN #GP(0); FI;
```

```
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) )
  THEN
    (* SS pages cannot be created on first or last page of ELRANGE *)
    IF ( TMP_LINADDR == DS:TMP_SECS.BASEADDR OR
        TMP_LINADDR == (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
      THEN
        #GP(0); FI;
```

```
FI;
```

```
(* Clear the content of EPC page*)
```

```
DS:RCX[32767:0] := 0;
```

```
(* Set EPCM security attributes *)
```

```
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
```

```
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
```

```
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
```

```
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
```

```
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
```

```
EPCM(DS:RCX).BLOCKED := 0;
```

```
EPCM(DS:RCX).PENDING := 1;
```

```
EPCM(DS:RCX).MODIFIED := 0;
```

```
EPCM(DS:RCX).PR := 0;
```

```
(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
```

```
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;
```

```
(* Set EPCM valid fields *)
```

```
EPCM(DS:RCX).VALID := 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked. If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked. If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

EBLOCK—Mark a page in EPC as Blocked

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLS[EBLOCK]	IR	V/V	SGX1	This leaf function marks a page in the EPC as blocked.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EBLOCK (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

EBLOCK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 40-12. EBLOCK Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EBLOCK successful.
SGX_BLKSTATE	Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).
SGX_ENTRYEPOCH_LOCKED	SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.
SGX_NOTBLOCKABLE	Page type is not one which can be blocked.
SGX_PG_INVLD	Page is not valid and cannot be blocked.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Table 40-13. Base Concurrency Restrictions of EBLOCK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EBLOCK	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Table 40-14. Additional Concurrency Restrictions of EBLOCK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EBLOCK	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EBLOCK Operational Flow

Name	Type	Size (Bits)	Description
TMP_BLKSTATE	Integer	64	Page is already blocked.

```
IF (DS:RCX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
  THEN #PF(DS:RCX); FI;
```

```
RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;
```

(* Check the EPC page for concurrency*)

```
IF (EPC page in use)
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_EPC_PAGE_CONFLICT;
    GOTO DONE;
```

```
FI;
```

```
IF (EPCM(DS:RCX).VALID = 0)
```

```
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_PG_INVLD;
    GOTO DONE;
```

```
FI;
```

```
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM)
and EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
```

```
  THEN
    RFLAGS.CF := 1;
    IF (EPCM(DS:RCX).PT = PT_SECS)
      THEN RAX := SGX_PG_IS_SECS;
    ELSE RAX := SGX_NOTBLOCKABLE;
```

```
  FI;
  GOTO DONE;
```

```
FI;
```

(* Check if the page is already blocked and report blocked state *)

```
TMP_BLKSTATE := EPCM(DS:RCX).BLOCKED;
```

```
(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)  
IF (TMP_BLKSTATE = 1)  
  THEN  
    RFLAGS.CF := 1;  
    RAX := SGX_BLKSTATE;  
  ELSE  
    EPCM(DS:RCX).BLOCKED := 1  
FI;  
DONE:
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the DS segment limit.
 If a memory operand is not properly aligned.
 If the specified EPC resource is in use.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If a memory operand is not an EPC page.

64-Bit Mode Exceptions

- #GP(0) If a memory operand is non-canonical form.
 If a memory operand is not properly aligned.
 If the specified EPC resource is in use.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If a memory operand is not an EPC page.

ECREATE—Create an SECS page in the Enclave Page Cache

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLS[ECREATE]	IR	V/V	SGX1	This leaf function begins an enclave build by creating an SECS page in EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

Description

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, ATTRIBUTES, CONFIGID and CONFIGSVN. SECS:BASEADDR must be naturally aligned on an SECS.SIZE boundary. SECS.SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

ECREATE Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAME-SIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP_SECS.ATTRIBUTES.XFRM size. Details of computing the size can be found Section 41.7.

Concurrency Restrictions

Table 40-15. Base Concurrency Restrictions of ECREATE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ECREATE	SECS [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 40-16. Additional Concurrency Restrictions of ECREATE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ECREATE	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ECREATE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the SECS source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.
TMP_XSIZE	SSA Size	64	The size calculation of SSA frame.
TMP_MISC_SIZE	MISC Field Size	64	Size of the selected MISC field components.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

IF (DS:RBX is not 32Byte Aligned)
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
 TMP_SECINFO := DS:RBX.SECINFO;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned)
 THEN #GP(0); FI;

IF (DS:RBX.LINADDR != 0 or DS:RBX.SECS != 0)
 THEN #GP(0); FI;

(* Check for misconfigured SECINFO flags*)

IF (DS:TMP_SECINFO reserved fields are not zero or DS:TMP_SECINFO.FLAGS.PT != PT_SECS)
 THEN #GP(0); FI;

TMP_SECS := RCX;

IF (EPC entry in use)
 THEN
 IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
 THEN
 VMCS.Exit_reason := SGX_CONFLICT;

```

        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address :=
            << translation of DS:TMP_SECS produced by paging >>;
        VMCS.Guest-linear_address := DS:TMP_SECS;
    Deliver VMEXIT;
    ELSE
        #GP(0);
FI;

FI;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPAGE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
    THEN #GP(0); FI;

(* Check legality of CET_ATTRIBUTES *)
IF ((DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_ATTRIBUTES ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[5:2] ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[1:0] ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1 and
    (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) not canonical) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0 and
    (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) & 0xFFFFFFFF00000000) ||
    (DS:TMP_SECS.CET_ATTRIBUTES.reserved fields not 0) or
    (DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) is not page aligned))
    THEN
        #GP(0);
FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( !(CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
    THEN
        EPCM(DS:TMP_SECS).EntryLock.Release();
        #GP(0);
FI;

(* Compute size of MISC area *)
TMP_MISC_SIZE := compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit, see Section 41.7.2.2*)

```


SGX INSTRUCTION REFERENCES

```
TMP_XSIZE := compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;
```

```
(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
```

```
IF ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFF00000000H) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
```

```
    THEN #GP(0); FI;
```

```
IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )
```

```
    THEN #GP(0); FI;
```

```
(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
```

```
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
```

```
    THEN #GP(0); FI;
```

```
(* Ensure base address of an enclave is aligned on size*)
```

```
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) ) )
```

```
    THEN #GP(0); FI;
```

```
(* Ensure the SECS does not have any unsupported attributes*)
```

```
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
```

```
    THEN #GP(0); FI;
```

```
IF ( DS:TMP_SECS reserved fields are not zero)
```

```
    THEN #GP(0); FI;
```

```
(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
```

```
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠ 0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0) )
```

```
    THEN #GP(0); FI;
```

```
Clear DS:TMP_SECS to Uninitialized;
```

```
DS:TMP_SECS.MRENCLAVE := SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
```

```
DS:TMP_SECS.ISVSVN := 0;
```

```
DS:TMP_SECS.ISVPRODID := 0;
```

```
(* Initialize hash updates etc*)
```

```
Initialize enclave's MRENCLAVE update counter;
```

```
(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
```

```
TMPUPDATEFIELD[63:0] := 0045544145524345H; // "ECREATE"
```

```
TMPUPDATEFIELD[95:64] := DS:TMP_SECS.SSAFRAMESIZE;
```

```
TMPUPDATEFIELD[159:96] := DS:TMP_SECS.SIZE;
```

```
IF (CPUID.(EAX=7, ECX=0):.EDX[CET_IBT] = 1)
```

```
    THEN
```

```
        TMPUPDATEFIELD[223:160] := DS:TMP_SECS.CET_LEG_BITMAP_OFFSET;
```

```
    ELSE
```

```
        TMPUPDATEFIELD[223:160] := 0;
```

```

FI;
TMPUPDATEFIELD[511:160] := 0;
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

```

(* Set EID *)

```
DS:TMP_SECS.EID := LockedXAdd(CR_NEXT_EID, 1);
```

(* Initialize the virtual child count to zero *)

```
DS:TMP_SECS.VIRTCHILDCNT := 0;
```

(* Load ENCLAVECONTEXT with Address out of paging of SECS *)

```
<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
```

(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)

```

EPCM(DS:TMP_SECS).PT := PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS := 0;
EPCM(DS:TMP_SECS).R := 0;
EPCM(DS:TMP_SECS).W := 0;
EPCM(DS:TMP_SECS).X := 0;

```

(* Set EPCM entry fields *)

```

EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).VALID := 1;

```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If the reserved fields are not zero. If PAGEINFO.SECS is not zero. If PAGEINFO.LINADDR is not zero. If the SECS destination is locked. If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If the SECS destination is outside the EPC.

64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory address is non-canonical form. If a memory operand is not properly aligned. If the reserved fields are not zero. If PAGEINFO.SECS is not zero. If PAGEINFO.LINADDR is not zero. If the SECS destination is locked. If SECS.SSAFRAMESIZE is insufficient.
--------	--

SGX INSTRUCTION REFERENCES

#PF(error code) If a page fault occurs in accessing memory operands.
 If the SECS destination is outside the EPC.

EDBGRD—Read From a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLS[EDBGRD]	IR	V/V	SGX1	This leaf function reads a dword/quadword from a debug enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGRD (In)	Return error code (Out)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGRD Memory Parameter Semantics

EPCQW
Read access permitted by Enclave

The error codes are:

Table 40-17. EDBGRD Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EDBGRD successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

The instruction faults if any of the following:

EDBGRD Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).
An operand causing any segment violation.	May page fault.
CPL > 0.	

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Table 40-18. Base Concurrency Restrictions of EDBGD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGD	Target [DS:RCX]	Shared	#GP	

Table 40-19. Additional Concurrency Restrictions of EDBGD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGD Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ((TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned))
 THEN #GP(0); FI;

IF ((TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned))
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing the same EPCM entry *)

IF (Another instruction modifying the same EPCM entry is executing)
 THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)
 THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (SOURCE) is pointing to a PT_REG or PT_TCS or PT_VA or PT_SS_FIRST or PT_SS_REST *)

IF ((EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_VA)
 and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
 THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0))
 THEN
 RFLAGS.ZF := 1;

```

    RAX := SGX_PAGE_NOT_DEBUGGABLE;
    GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX).PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
    THEN
        TMP_SECS := GET_SECS_ADDRESS;
        IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
            THEN #GP(0); FI;
        IF ( (TMP_MODE64 = 1) )
            THEN RBX[63:0] := (DS:RCX)[63:0];
            ELSE EBX[31:0] := (DS:RCX)[31:0];
        FI;
    ELSE
        TMP_64BIT_VAL[63:0] := (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
        IF (TMP_MODE64 = 1)
            THEN
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN RBX[63:0] := 0FFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] := 0H;
                FI;
            ELSE
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN EBX[31:0] := 0FFFFFFFFFH;
                    ELSE EBX[31:0] := 0H;
                FI;
            FI;
    FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;

```

Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights. If DS segment is unusable. If RCX points to a memory location not 4Byte-aligned. If the address in RCX points to a page belonging to a non-debug enclave. If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA. If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
--------	---

SGX INSTRUCTION REFERENCES

#PF(error code) If a page fault occurs in accessing memory operands.
 If the address in RCX points to a non-EPC page.
 If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0) If RCX is non-canonical form.
 If RCX points to a memory location not 8Byte-aligned.
 If the address in RCX points to a page belonging to a non-debug enclave.
 If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
 If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.

#PF(error code) If a page fault occurs in accessing memory operands.
 If the address in RCX points to a non-EPC page.
 If the address in RCX points to an invalid EPC page.

EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGWR (In)	Return error code (Out)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGWR Memory Parameter Semantics

EPCQW
Write access permitted by Enclave

The instruction faults if any of the following:

EDBGWR Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

Table 40-20. EDBGWR Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EDBGWR successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGWR does not result in a #GP.

Concurrency Restrictions

Table 40-21. Base Concurrency Restrictions of EDBGWR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGWR	Target [DS:RCX]	Shared	#GP	

Table 40-22. Additional Concurrency Restrictions of EDBGWR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGWR	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGWR Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ((TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned))
 THEN #GP(0); FI;

IF ((TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned))
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing the same EPCM entry *)

IF (Another instruction modifying the same EPCM entry is executing)
 THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)
 THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)

IF ((EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
 and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
 THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)

IF ((EPCM(DS:RCX).PENDING is not 0) or (EPCM(DS:RCX).MODIFIED is not 0))
 THEN
 RFLAGS.ZF := 1;

```

    RAX := SGX_PAGE_NOT_DEBUGGABLE;
    GOTO DONE;
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX).PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & OFF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS := GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] := RBX[63:0];
    ELSE (DS:RCX)[31:0] := EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0

```

Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If the address in RCS violates DS limit or access rights. If DS segment is unusable. If RCX points to a memory location not 4Byte-aligned. If the address in RCX points to a page belonging to a non-debug enclave. If the address in RCX points to a page which is not PT_TCS or PT_REG. If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If the address in RCX points to a non-EPC page. If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If RCX is non-canonical form. If RCX points to a memory location not 8Byte-aligned. If the address in RCX points to a page belonging to a non-debug enclave. If the address in RCX points to a page which is not PT_TCS or PT_REG. If the address in RCX points to a location inside TCS that is not the FLAGS word.
--------	--

SGX INSTRUCTION REFERENCES

#PF(error code) If a page fault occurs in accessing memory operands.
 If the address in RCX points to a non-EPC page.
 If the address in RCX points to an invalid EPC page.

EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized enclave page.

Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string comprising of “EEXTEND” || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

The instruction faults if any of the following:

EEXTEND Faulting Conditions

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

Concurrency Restrictions

Table 40-23. Base Concurrency Restrictions of EEXTEND

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXTEND	Target [DS:RCX]	Shared	#GP	
	SECS [DS:RBX]	Concurrent		

Table 40-24. Additional Concurrency Restrictions of EEXTEND

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXTEND	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EEXTEND Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFS ET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)
THEN #GP(0); FI;

IF (DS:RBX does resolve to an EPC page)
THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other instructions accessing EPCM)
THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ((EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
THEN #PF(DS:RCX); FI;

TMP_SECS := Get_SECS_ADDRESS();

IF (DS:RBX does not resolve to TMP_SECS)
THEN #GP(0); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ((Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))

```
THEN #GP(0); FI;
```

```
(* Calculate enclave offset *)
```

```
TMP_ENCLAVEOFFSET := EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
```

```
TMP_ENCLAVEOFFSET := TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)
```

```
(* Add EEXTEND message and offset to MRENCLAVE *)
```

```
TMPUPDATEFIELD[63:0] := 00444E4554584545H; // "EEXTEND"
```

```
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
```

```
TMPUPDATEFIELD[511:128] := 0; // 48 bytes
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
```

```
INC enclave's MRENCLAVE update counter;
```

```
(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
```

```
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
```

```
INC enclave's MRENCLAVE update counter by 4;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If the address in RBX is outside the DS segment limit. If RBX points to an SECS page which is not the SECS of the data chunk. If the address in RCX is outside the DS segment limit. If RCX points to a memory location not 256Byte-aligned. If another instruction is accessing MRENCLAVE. If another instruction is checking or updating the SECS. If the enclave is already initialized.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If the address in RBX points to a non-EPC page. If the address in RCX points to a page which is not PT_TCS or PT_REG. If the address in RCX points to a non-EPC page. If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If RBX is non-canonical form. If RBX points to an SECS page which is not the SECS of the data chunk. If RCX is non-canonical form. If RCX points to a memory location not 256 Byte-aligned. If another instruction is accessing MRENCLAVE. If another instruction is checking or updating the SECS. If the enclave is already initialized.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If the address in RBX points to a non-EPC page. If the address in RCX points to a page which is not PT_TCS or PT_REG. If the address in RCX points to a non-EPC page. If the address in RCX points to an invalid EPC page.

EINIT—Initialize an Enclave for Execution

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLS[EINIT]	IR	V/V	SGX1	This leaf function initializes the enclave and makes it ready to execute enclave code.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

Description

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

$$q1 = \text{floor}(\text{Signature}^2 / \text{Modulus});$$

$$q2 = \text{floor}((\text{Signature}^3 - q1 * \text{Signature} * \text{Modulus}) / \text{Modulus});$$

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

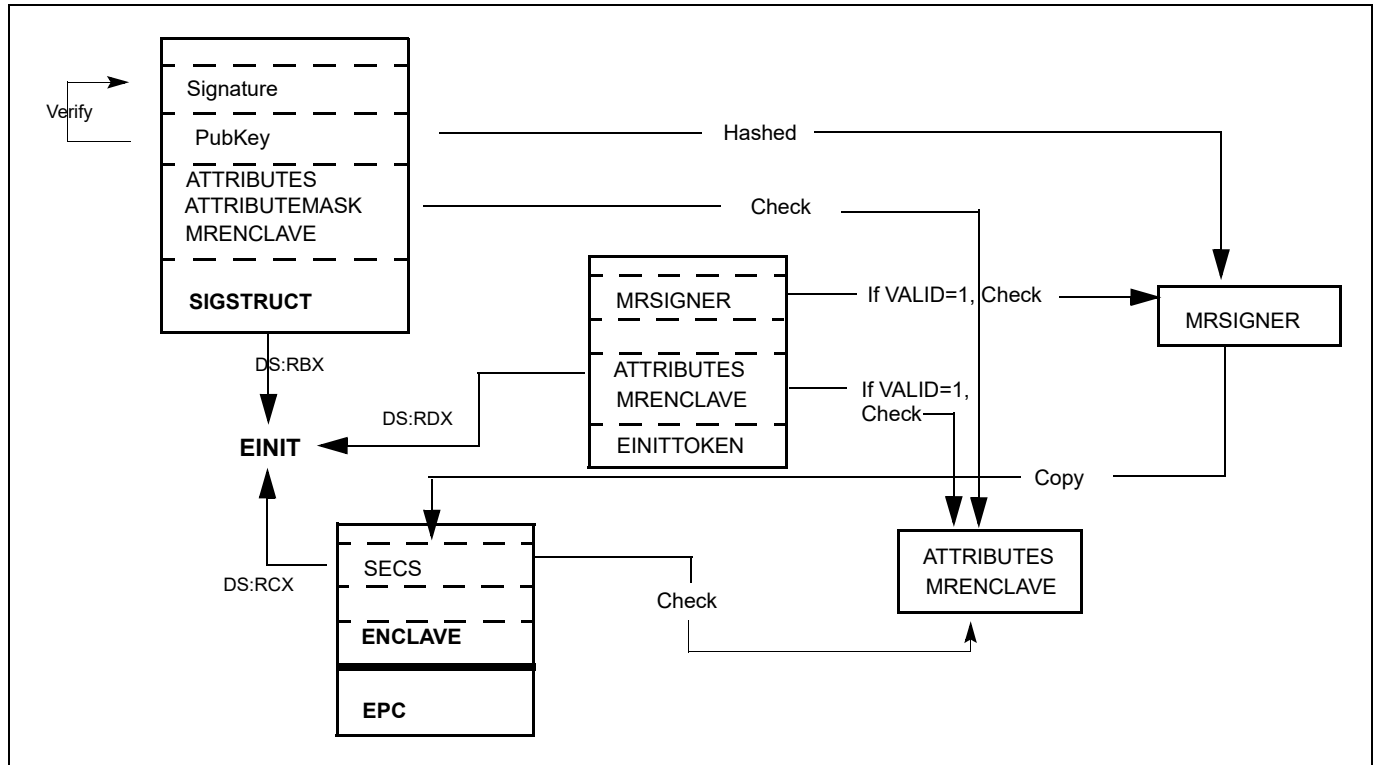


Figure 40-1. Relationships Between SECS, SIGSTRUCT and EINITTOKEN

EINIT Memory Parameter Semantics

SIGSTRUCT	SECS	EINITOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps, which can be seen in Figure 40-1:

Validates that SIGSTRUCT is signed using the enclosed public key.

Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.

Checks that no reserved bits are set to 1 in SIGSTRUCT.ATTRIBUTES and no reserved bits in SIGSTRUCT.ATTRIBUTESMASK are set to 0.

Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

Checks that SIGSTRUCT.ATTRIBUTES equals the result of logically and-ing SIGSTRUCT.ATTRIBUTESMASK with SECS.ATTRIBUTES.

If EINITOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

If EINITOKEN.VALID is 1, checks the validity of EINITOKEN.

If EINITOKEN.VALID is 1, checks that EINITOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITOKEN.VALID is 1 and EINITOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.

Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.

Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX_UNMASKED_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

Table 40-25. EINIT Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EINIT successful.
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value.
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask.
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement. If EINITOKEN contains an incorrect measurement.
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key.
SGX_INVALID_LICENSE	If license is invalid.
SGX_INVALID_CPUSVN	If license SVN is unsupported.
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation.

Concurrency Restrictions

Table 40-26. Base Concurrency Restrictions of EINIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS [DS:RCX]	Shared	#GP	

Table 40-27. Additional Concurrency Restrictions of ENIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINIT	SECS [DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature ³ modulo MRSIGNER.

(* make sure SIGSTRUCT and SECS are aligned *)
 IF ((DS:RBX is not 4KByte Aligned) or (DS:RCX is not 4KByte Aligned))
 THEN #GP(0); FI;

(* make sure the EINITTOKEN is aligned *)
 IF (DS:RDX is not 512Byte Aligned)
 THEN #GP(0); FI;

(* make sure the SECS is inside the EPC *)
 IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

TMP_SIG[14463:0] := DS:RBX[14463:0]; // 1808 bytes
 TMP_TOKEN[2423:0] := DS:RDX[2423:0]; // 304 bytes

```

(* Verify SIGSTRUCT Header. *)
IF ( (TMP_SIG.HEADER ≠ 06000000E100000000001000000000h) or
    ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
    (TMP_SIG.HEADER2 ≠ 0101000060000000600000001000000h) or
    (TMP_SIG.EXPONENT ≠ 00000003h) or (Reserved space is not 0's) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* Open "Event Window" Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_UNMASKED_EVENT;
    GOTO EXIT;
FI

IF (signature failed to verify) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_SIGNATURE;
    GOTO EXIT;
FI;

(*Close "Event Window" *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS)
    THEN #GP(0); FI;

IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify ISVFAMILYID is not used on an enclave with KSS disabled *)
IF ((TMP_SIG.ISVFAMILYID != 0) AND (DS:RCX.ATTRIBUTES.KSS == 0))
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS's Initialized state))
    THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE := SHA256FINAL( (DS:RCX).MRENCLAVE, enclave's MRENCLAVE update count *512);

(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

```

```
TMP_MRSIGNER := SHA256(TMP_SIG.MODULUS)
```

```
(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
```

```
CONTROLLED_ATTRIBUTES := 000000000000020H;
```

```
IF ( ( DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES ) ≠ 0 ) and ( TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
```

```
IF ( ( DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK ) ≠ ( TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(*Verify SIGSTRUCT.MISCSELECT requirements are met *)
```

```
IF ( ( DS:RCX.MISCSELECT & TMP_SIG.MISCMASK ) ≠ ( TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK ) )
```

```
    THEN
```

```
        RFLAGS.ZF := 1;
```

```
        RAX := SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT
```

```
FI;
```

```
IF ( CPUID.(EAX=12H, ECX=1):EAX[6] = 1 )
```

```
    IF ( DS:RCX.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIBUTES_MASK ≠ TMP_SIG.CET_ATTRIBUTES &
```

```
        TMP_SIG.CET_ATTRIBUTES_MASK )
```

```
        THEN
```

```
            RFLAGS.ZF := 1;
```

```
            RAX := SGX_INVALID_ATTRIBUTE;
```

```
            GOTO EXIT
```

```
    FI;
```

```
FI;
```

```
(* If EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
```

```
IF ( TMP_TOKEN.VALID[0] = 0 )
```

```
    IF ( TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH )
```

```
        RFLAGS.ZF := 1;
```

```
        RAX := SGX_INVALID_EINITTOKEN;
```

```
        GOTO EXIT;
```

```
    FI;
```

```
    GOTO COMMIT;
```

```
FI;
```

```
(* Debug Launch Enclave cannot launch Production Enclaves *)
```

```
IF ( ( DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1 ) and ( DS:RCX.ATTRIBUTES.DEBUG = 0 ) )
```

```
    RFLAGS.ZF := 1;
```

```
    RAX := SGX_INVALID_EINITTOKEN;
```

```
    GOTO EXIT;
```

```
FI;
```

(* Check reserve space in EINIT token includes reserved regions and upper bits in valid field *)

IF (TMP_TOKEN.reserved_space_is_not_clear)

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_EINITTOKEN;
GOTO EXIT;
```

FI;

(* EINIT token must not have been created by a configuration beyond the current CPU configuration *)

IF (TMP_TOKEN.CPUSVN must not be a configuration beyond CR_CPUSVN)

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_CPUSVN;
GOTO EXIT;
```

FI;

(* Derive Launch key used to calculate EINITTOKEN.MAC *)

```
HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;
```

```
TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN := TMP_TOKEN.ISVSVNLE;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID := TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := TMP_TOKEN.CPUSVNLE;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_TOKEN.CET_MASKED_ATTRIBUTES_LE;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
```

FI;

(* Calculate the derived key*)

```
TMP_EINITTOKENKEY := derivekey(TMP_KEYDEPENDENCIES);
```

(* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch Enclave. Only 192 bytes of EINITTOKEN are CMACed *)

IF (TMP_TOKEN.MAC ≠ CMAC(TMP_EINITTOKENKEY, TMP_TOKEN[1535:0]))

```
RFLAGS.ZF := 1;
RAX := SGX_INVALID_EINITTOKEN;
GOTO EXIT;
```

FI;

SGX INSTRUCTION REFERENCES

```
(* Verify EINITTOKEN (RDX) is for this enclave *)
IF ( (TMP_TOKEN.MRENCLAVE ≠ TMP_MRENCLAVE) or (TMP_TOKEN.MRSIGNER ≠ TMP_MRSIGNER) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

(* Verify ATTRIBUTES in EINITTOKEN are the same as the enclave's *)
IF (TMP_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINIT_ATTRIBUTE;
    GOTO EXIT;
FI;

COMMIT:
(* Commit changes to the SECS; Set ISVPRODID, ISVSVN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) *)
DS:RCX.MRENCLAVE := TMP_MRENCLAVE;
(* MRSIGNER stores a SHA256 in little endian implemented natively on x86 *)
DS:RCX.MRSIGNER := TMP_MRSIGNER;
DS:RCX.ISVEXTPRODID := TMP_SIG.ISVEXTPRODID;
DS:RCX.ISVPRODID := TMP_SIG.ISVPRODID;
DS:RCX.ISVSVN := TMP_SIG.ISVSVN;
DS:RCX.ISVFAMILYID := TMP_SIG.ISVFAMILYID;
DS:RCX.PADDING := TMP_SIG.PADDING;

(* Mark the SECS as initialized *)
Update DS:RCX to initialized;

(* Set RAX and ZF for success*)
    RFLAGS.ZF := 0;
    RAX := 0;
EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands. If RCX does not resolve in an EPC page. If the memory address is not a valid, uninitialized SECS.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
--------	---

#PF(error code) If a page fault occurs in accessing memory operands.
If RCX does not resolve in an EPC page.
If the memory address is not a valid, uninitialized SECS.

ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.
EAX = 12H ENCLS[ELDBC]	IR	V/V	EAX[6]	This leaf function behaves like ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELDUC]	IR	V/V	EAX[6]	This leaf function behaves like ELDU but with improved conflict handling for oversubscription.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	ELDB/ELDU (In)	Return error code (Out)	Address of the PAGEINFO (In)	Address of the EPC page (In)	Address of the version- array slot (In)

Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The ELDBC/ELDUC leafs are very similar to ELDB and ELDU. They provide an error code on the concurrency conflict for any of the pages which need to acquire a lock. These include the destination, SECS, and VA slot.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

ELDB/ELDU/ELDBC/ELBUC Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	PAGEINFO.SECS	EPCPAGE	Version-Array Slot
Non-enclave read access	Non-enclave read access	Non-enclave read access	Enclave read/write access	Read/Write access permitted by Enclave	Read/Write access per- mitted by Enclave

The error codes are:

Table 40-28. ELDB/ELDU/ELDBC/ELBUC Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	ELDB/ELDU successful.
SGX_MAC_COMPARE_FAIL	If the MAC check fails.

Concurrency Restrictions

Table 40-29. Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ELDB/ELDU	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	
ELDBC/ELBUC	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA [DS:RDX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	SGX_EPC_PAGE_CONFLICT	

Table 40-30. Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ELDB/ELDU	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	
ELDBC/ELBUC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ELDB/ELDU/ELDBC/ELBUC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Memory page	4KBytes	
TMP_SECS	Memory page	4KBytes	
TMP_PCMD	PCMD	128 Bytes	
TMP_HEADER	MACHEADER	128 Bytes	
TMP_VER	UINT64	64	
TMP_MAC	UINT128	128	
TMP_PK	UINT128	128	Page encryption/MAC key.
SCRATCH_PCMD	PCMD	128 Bytes	

(* Check PAGEINFO and EPCPAGE alignment *)

IF ((DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned))
THEN #GP(0); FI;

SGX INSTRUCTION REFERENCES

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

```
(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;
```

```
TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_PCMD := DS:RBX.PCMD;
```

```
(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;
```

```
(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            THEN
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason := SGX_CONFLICT;
                        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                        VMCS.Exit_qualification.error := 0;
                        VMCS.Guest-physical_address :=
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address := DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason := SGX_CONFLICT;
                        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_ERROR;
                        VMCS.Exit_qualification.error := SGX_EPC_PAGE_CONFLICT;
                        VMCS.Guest-physical_address :=
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address := DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        RFLAGS.ZF := 1;
                        RFLAGS.CF := 0;
                        RAX := SGX_EPC_PAGE_CONFLICT;
                        GOTO ERROR_EXIT;
                FI;
```

```

    FI;
FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            #GP(0);
            FI;
        ELSE (* ELDBC/ELDUC *)
            RFLAGS.ZF := 1;
            RFLAGS.CF := 0;
            RAX := SGX_EPC_PAGE_CONFLICT;
            GOTO ERROR_EXIT;
        FI;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~OFFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0] := DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0] := 0;

TMP_HEADER.SECINFO := SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD := SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR := DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) )
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Other instructions modifying SECS)
            THEN
                IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
                    #GP(0);
                    FI;
                ELSE (* ELDBC/ELDUC *)
                    RFLAGS.ZF := 1;
                    RFLAGS.CF := 0;
                    RAX := SGX_EPC_PAGE_CONFLICT;
                    GOTO ERROR_EXIT;
                FI;
        FI;
FI;

```

SGX INSTRUCTION REFERENCES

```
FI;

IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    THEN
        TMP_HEADER.EID := DS:TMP_SECS.EID;
    ELSE
        (* These pages do not have any parent, and hence no EID binding *)
        TMP_HEADER.EID := 0;
FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767: 0] := DS:TMP_SRCPGE[32767: 0];
TMP_VER := DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} := AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_MAC_COMPARE_FAIL;
        GOTO ERROR_EXIT;
FI;

(* Check version before committing *)
IF (DS:RDX ≠ 0)
    THEN #GP(0);
    ELSE
        DS:RDX := TMP_VER;
FI;

(* Commit EPCM changes *)
EPCM(DS:RCX).PT := TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX := TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING := TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED := TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR := TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
    THEN
        EPCM(DS:RCX).BLOCKED := 1;
    ELSE
        EPCM(DS:RCX).BLOCKED := 0;
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
    << store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;
```

EPCM(DS:RCX). VALID := 1;

RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If the instruction's EPC resource is in use by others. If the instruction fails to verify MAC. If the version-array slot is in use. If the parameters fail consistency checks.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If a memory operand expected to be in EPC does not resolve to an EPC page. If one of the EPC memory operands has incorrect page type. If the destination EPC page is already valid.

64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory operand is non-canonical form. If a memory operand is not properly aligned. If the instruction's EPC resource is in use by others. If the instruction fails to verify MAC. If the version-array slot is in use. If the parameters fail consistency checks.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If a memory operand expected to be in EPC does not resolve to an EPC page. If one of the EPC memory operands has incorrect page type. If the destination EPC page is already valid.

EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODPR Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 40-31. EMODPR Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EMODPR successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Table 40-32. Base Concurrency Restrictions of EMODPR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPR	Target [DS:RCX]	Shared	#GP	

Table 40-33. Additional Concurrency Restrictions of EMODPR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)

IF ((SCRATCH_SECINFO reserved fields are not zero) or
(SCRATCH_SECINFO.FLAGS.R is 0 and SCRATCH_SECINFO.FLAGS.W is not 0))
THEN #GP(0); FI;

(* Check concurrency with SGX1 or SGX2 instructions on the EPC page *)
IF (SGX1 or other SGX2 instructions accessing EPC page)
THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0)
THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
THEN
RFLAGS.ZF := 1;
RAX := SGX_EPC_PAGE_CONFLICT;
GOTO DONE;

FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0))
THEN
RFLAGS.ZF := 1;
RAX := SGX_PAGE_NOT_MODIFIABLE;

SGX INSTRUCTION REFERENCES

```
GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR := 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R := EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODT Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 40-34. EMODT Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EMODT successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.

Concurrency Restrictions

Table 40-35. Base Concurrency Restrictions of EMODT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR

Table 40-36. Additional Concurrency Restrictions of EMODT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)

IF ((SCRATCH_SECINFO reserved fields are not zero) or
 !(SCRATCH_SECINFO.FLAGS.PT is PT_TCS or SCRATCH_SECINFO.FLAGS.PT is PT_TRIM))
 THEN #GP(0); FI;

(* Check concurrency with SGX1 instructions on the EPC page *)

IF (other SGX1 instructions accessing EPC page)
 THEN
 RFLAGS.ZF := 1;
 RAX := SGX_EPC_PAGE_CONFLICT;
 GOTO DONE;

FI;

IF (EPCM(DS:RCX).VALID is 0)
 THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)

IF (EPC page in use by another SGX2 instruction)
 THEN
 RFLAGS.ZF := 1;
 RAX := SGX_EPC_PAGE_CONFLICT;
 GOTO DONE;

```

FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
    ((EPCM(DS:RCX).PT is PT_TCS or PT_SS_FIRST or PT_SS_REST) and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).MODIFIED := 1;
EPCM(DS:RCX).R := 0;
EPCM(DS:RCX).W := 0;
EPCM(DS:RCX).X := 0;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

EPA—Add Version Array

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0AH ENCLS[EPA]	IR	V/V	SGX1	This leaf function adds a Version Array to the EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EPA (In)	PT_VA (In, Constant)	Effective address of the EPC page (In)

Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

EPA Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

Concurrency Restrictions

Table 40-37. Base Concurrency Restrictions of EPA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EPA	VA [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 40-38. Additional Concurrency Restrictions of EPA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EPA	VA [DS:RCX]	Concurrent	L	Concurrent		Concurrent	

Operation

IF (RBX ≠ PT_VA or DS:RCX is not 4KByte Aligned)
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)

IF (Other Intel SGX instructions accessing the page)
THEN

IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)

```

THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
    VMCS.Guest-linear_address := DS:RCX;
    Deliver VMEXIT;
ELSE
    #GP(0);
FI;
FI;

```

(* Check EPC page must be empty *)

```

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

```

(* Clears EPC page *)

```

DS:RCX[32767:0] := 0;

```

```

EPCM(DS:RCX).PT := PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS := 0;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).RWX := 0;
EPCM(DS:RCX).VALID := 1;

```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If a memory operand is not properly aligned.</p> <p>If another Intel SGX instruction is accessing the EPC page.</p> <p>If RBX is not set to PT_VA.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If a memory operand is not an EPC page.</p> <p>If the EPC page is valid.</p>

64-Bit Mode Exceptions

#GP(0)	<p>If a memory operand is non-canonical form.</p> <p>If a memory operand is not properly aligned.</p> <p>If another Intel SGX instruction is accessing the EPC page.</p> <p>If RBX is not set to PT_VA.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If a memory operand is not an EPC page.</p> <p>If the EPC page is valid.</p>

ERDINFO—Read Type and Status Information About an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 10H ENCLS[ERDINFO]	IR	V/V	EAX[6]	This leaf function returns type and status information about an EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	ERDINFO (In)	Return error code (Out)	Address of a RDINFO structure (In)	Address of the destination EPC page (In)

Description

This instruction reads type and status information about an EPC page and returns it in a RDINFO structure. The STATUS field of the structure describes the status of the page and determines the validity of the remaining fields. The FLAGS field returns the EPCM permissions of the page; the page type; and the BLOCKED, PENDING, MODIFIED, and PR status of the page. For enclave pages, the ENCLAVECONTEXT field of the structure returns the value of SECS.ENCLAVECONTEXT. For non-enclave pages (e.g., VA) ENCLAVECONTEXT returns 0.

For invalid or non-EPC pages, the instruction returns an information code indicating the page's status, in addition to populating the STATUS field.

ERDINFO returns an error code if the destination EPC page is being modified by a concurrent SGX instruction.

RBX contains the effective address of a RDINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of ERDINFO leaf function.

ERDINFO Memory Parameter Semantics

RDINFO	EPCPAGE
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

ERDINFO Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

The error codes are:

Table 40-39. ERDINFO Return Value in RAX

Error Code	Value	Description
No Error	0	ERDINFO successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD		Target page is not a valid EPC page.
SGX_PG_NONEPC		Page is not an EPC page.

Concurrency Restrictions

Table 40-40. Base Concurrency Restrictions of ERDINFO

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERDINFO	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Table 40-41. Additional Concurrency Restrictions of ERDINFO

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERDINFO Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_RDINFO	Linear Address	64	Address of the RDINFO structure.

(* check alignment of RDINFO structure (RBX) *)
 IF (DS:RBX is not 32Byte Aligned) THEN
 #GP(0); FI;

(* check alignment of the EPCPAGE (RCX) *)
 IF (DS:RCX is not 4KByte Aligned) THEN
 #GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
 IF (DS:RCX does not resolve within EPC) THEN
 RFLAGS.CF := 1;
 RFLAGS.ZF := 0;
 RAX := SGX_PG_NONEPC;
 goto DONE;
 FI;

(* Check the EPC page for concurrency *)
 IF (EPC page is being modified) THEN
 RFLAGS.ZF = 1;
 RFLAGS.CF = 0;
 RAX = SGX_EPC_PAGE_CONFLICT;
 goto DONE;
 FI;

(* check page validity *)
 IF (EPCM(DS:RCX).VALID = 0) THEN
 RFLAGS.CF = 1;

SGX INSTRUCTION REFERENCES

```
RFLAGS.ZF = 0;
RAX = SGX_PG_INVLD;
goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO := DS:RBX;
TMP_RDINFO.STATUS := 0;
TMP_RDINFO.FLAGS := 0;
TMP_RDINFO.ENCLAVECONTEXT := 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_RDINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR := EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE := EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED := EPCM(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_REST)
) THEN
    TMP_SECS := Address of SECS for (DS:RCX);
    TMP_RDINFO.ENCLAVECONTEXT := SECS(TMP_SECS).ENCLAVECONTEXT;
FI;

(* populate enclave information for SECS pages *)
IF (EPCM(DS:RCX).PAGE_TYPE = PT_SECS) THEN
    IF ((VMX non-root mode) and
        (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)
    ) THEN
        TMP_RDINFO.STATUS.CHILDPRESENT :=
            ((SECS(DS:RCX).CHLDCNT ≠ 0) or
             SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
    ELSE
        TMP_RDINFO.STATUS.CHILDPRESENT := (SECS(DS:RCX).CHLDCNT ≠ 0);
        TMP_RDINFO.STATUS.VIRTCHILDPRESENT :=
            (SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
        TMP_RDINFO.ENCLAVECONTEXT := SECS(DS:RCX).ENCLAVECONTEXT;
    FI;
FI;

RAX := 0;
RFLAGS.ZF := 0;
RFLAGS.CF := 0;

DONE:
(* clear flags *)
RFLAGS.PF := 0;
RFLAGS.AF := 0;
```

RFLAGS.OF := 0;
RFLAGS.SF := ? 0;

Flags Affected

ZF is set if ERDINFO fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF is set if page is not a valid EPC page or not an EPC page; otherwise cleared.

PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the DS segment limit.
 If DS segment is unusable.
 If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
 If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.

EREMOVE—Remove a page from the EPC

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLS[EREMOVE]	IR	V/V	SGX1	This leaf function removes a page from the EPC.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EREMOVE (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

EREMOVE Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

The instruction faults if any of the following:

EREMOVE Faulting Conditions

The memory operand is not properly aligned.	The memory operand does not resolve in an EPC page.
Refers to an invalid SECS.	Refers to an EPC page that is locked by another thread.
Another Intel SGX instruction is accessing the EPC page.	RCX does not contain an effective address of an EPC page.
the EPC page refers to an SECS with associations.	

The error codes are:

Table 40-42. EREMOVE Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EREMOVE successful.
SGX_CHILD_PRESENT	If the SECS still have enclave pages loaded into EPC.
SGX_ENCLAVE_ACT	If there are still logical processors executing inside the enclave.

Concurrency Restrictions

Table 40-43. Base Concurrency Restrictions of EREMOVE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREMOVE	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 40-44. Additional Concurrency Restrictions of EREMOVE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREMOVE	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREMOVE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.

```
IF (DS:RCX is not 4KByte Aligned)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve to an EPC page)
  THEN #PF(DS:RCX); FI;
```

```
TMP_SECS := Get_SECS_ADDRESS();
```

```
(* Check the EPC page for concurrency *)
```

```
IF (EPC page being referenced by another Intel SGX instruction)
```

```
  THEN
```

```
    IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
```

```
      THEN
```

```
        VMCS.Exit_reason := SGX_CONFLICT;
```

```
        VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
```

```
        VMCS.Exit_qualification.error := 0;
```

```
        VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
```

```
        VMCS.Guest-linear_address := DS:RCX;
```

```
        Deliver VMEXIT;
```

```
      ELSE
```

```
        #GP(0);
```

```
    FI;
```

```
FI;
```

```
(* if DS:RCX is already unused, nothing to do*)
```

```
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
```

```
  THEN GOTO DONE;
```

```
FI;
```

SGX INSTRUCTION REFERENCES

```
IF ( (EPCM(DS:RCX).PT = PT_VA) OR
      ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
  THEN
    EPCM(DS:RCX).VALID := 0;
    GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
  THEN
    IF (DS:RCX has an EPC page associated with it)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_CHILD_PRESENT;
        GOTO ERROR_EXIT;
    FI;
    (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
    IF (<<in VMX non-root operation>> AND
        <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
        (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_CHILD_PRESENT;
        GOTO ERROR_EXIT;
    FI;
    EPCM(DS:RCX).VALID := 0;
    GOTO DONE;
FI;

IF (Other threads active using SECS)
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_ENCLAVE_ACT;
    GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
      (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
  THEN
    EPCM(DS:RCX).VALID := 0;
    GOTO DONE;
FI;

DONE:
RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the DS segment limit.
 If a memory operand is not properly aligned.
 If another Intel SGX instruction is accessing the page.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If the memory operand is not an EPC page.

64-Bit Mode Exceptions

- #GP(0) If the memory operand is non-canonical form.
 If a memory operand is not properly aligned.
 If another Intel SGX instruction is accessing the page.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If the memory operand is not an EPC page.

ETRACK—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0CH ENCLS[ETRACK]	IR	V/V	SGX1	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	ETRACK (In)	Return error code (Out)	Pointer to the SECS of the EPC page (In)

Description

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 40-45. ETRACK Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	ETRACK successful.
SGX_PREV_TRK_INCMPL	All processors did not complete the previous shoot-down sequence.

Concurrency Restrictions

Table 40-46. Base Concurrency Restrictions of ETRACK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACK	SECS [DS:RCX]	Shared	#GP	

Table 40-47. Additional Concurrency Restrictions of ETRACK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACK	SECS [DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

```

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
                Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
    FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PT ≠ PT_SECS)
    THEN #PF(DS:RCX); FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (DS:RCX).TRACKING ≠ 0 )
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
                Deliver VMEXIT;
            FI;
        RFLAGS.ZF := 1;
        RAX := SGX_PREV_TRK_INCMPL;
        GOTO DONE;
    ELSE
        RAX := 0;
        RFLAGS.ZF := 0;
    FI;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the DS segment limit.
 If a memory operand is not properly aligned.
 If another thread is concurrently using the tracking facility on this SECS.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If a memory operand is not an EPC page.

64-Bit Mode Exceptions

- #GP(0) If a memory operand is non-canonical form.
 If a memory operand is not properly aligned.
 If the specified EPC resource is in use.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If a memory operand is not an EPC page.

ETRACKC—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 11H ENCLS[ETRACKC]	IR	V/V	EAX[6]	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX	
IR	ETRACK (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Address of the SECS page (In, EA)

Description

The ETRACKC instruction is thread safe variant of ETRACK leaf and can be executed concurrently with other CPU threads operating on the same SECS.

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACKC Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 40-48. ETRACKC Return Value in RAX

Error Code	Value	Description
No Error	0	ETRACKC successful.
SGX_EPC_PAGE_CONFLICT	7	Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD	6	Target page is not a VALID EPC page.
SGX_PREV_TRK_INCMPL	17	All processors did not complete the previous tracking sequence.
SGX_TRACK_NOT_REQUIRED	27	Target page type does not require tracking.

Concurrency Restrictions

Table 40-49. Base Concurrency Restrictions of ETRACKC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACKC	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS implicit	Concurrent		

Table 40-50. Additional Concurrency Restrictions of ETRACKC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACKC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

Temp Variables in ETRACKC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

(* check alignment of EPCPAGE (RCX) *)
 IF (DS:RCX is not 4KByte Aligned) THEN
 #GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
 IF (DS:RCX does not resolve within an EPC) THEN
 #PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPC page for concurrency *)
 IF (EPC page is being modified) THEN
 RFLAGS.ZF := 1;
 RFLAGS.CF := 0;
 RAX := SGX_EPC_PAGE_CONFLICT;
 goto DONE_POST_LOCK_RELEASE;
 FI;

(* check to make sure the page is valid *)
 IF (EPCM(DS:RCX).VALID = 0) THEN
 RFLAGS.ZF := 1;
 RFLAGS.CF := 0;
 RAX := SGX_PG_INVLD;
 GOTO DONE;
 FI;

(* find out the target SECS page *)
 IF (EPCM(DS:RCX).PT is PT_REG or PT_TCS or PT_TRIM or PT_SS_FIRST or PT_SS_REST) THEN
 TMP_SECS := Obtain SECS through EPCM(DS:RCX).ENCLAVESECS;
 ELSE IF (EPCM(DS:RCX).PT is PT_SECS) THEN
 TMP_SECS := Obtain SECS through (DS:RCX);
 ELSE
 RFLAGS.ZF := 0;
 RFLAGS.CF := 1;
 RAX := SGX_TRACK_NOT_REQUIRED;
 GOTO DONE;
 FI;

```
(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS) THEN
  IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address :=
      SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_EPC_PAGE_CONFLICT;
  GOTO DONE;
```

```
FI;
(* All processors must have completed the previous tracking cycle*)
IF ( (TMP_SECS).TRACKING ≠ 0 )
THEN
  IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit_reason := SGX_CONFLICT;
    VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
    VMCS.Exit_qualification.error := 0;
    VMCS.Guest-physical_address :=
      SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_PREV_TRK_INCMPL;
  GOTO DONE;
```

```
FI;

RFLAGS.ZF := 0;
RFLAGS.CF := 0;
RAX := 0;
```

```
DONE:
(* clear flags *)
RFLAGS.PF,AF,OF,SF := 0;
```

Flags Affected

ZF is set if ETRACKC fails due to concurrent operations with another SGX instructions or target page is an invalid EPC page or tracking is not completed on SECS page; otherwise cleared.

CF is set if target page is not of a type that requires tracking; otherwise cleared.

PF, AF, OF and SF are cleared.

Protected Mode Exceptions

- #GP(0) If the memory operand violates access-control policies of DS segment.
 If DS segment is unusable.
- #PF(error code) If the memory operand is not properly aligned.
 If the memory operand expected to be in EPC does not resolve to an EPC page.
 If a page fault occurs in access memory operand.

64-Bit Mode Exceptions

- #GP(0) If a memory address is in a non-canonical form.
- #PF(error code) If a memory operand is not properly aligned.
 If the memory operand expected to be in EPC does not resolve to an EPC page.
 If a page fault occurs in access memory operand.

EWB—Invalidate an EPC Page and Write out to Main Memory

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0BH ENCLS[EWB]	IR	V/V	SGX1	This leaf function invalidates an EPC page and writes it out to main memory.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EWB (In)	Error code (Out)	Address of an PAGEINFO (In)	Address of the EPC page (In)	Address of a VA slot (In)

Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

EWB Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	EPCPAGE	VASLOT
Non-EPC R/W access	Non-EPC R/W access	Non-EPC R/W access	EPC R/W access	EPC R/W access

The error codes are:

Table 40-51. EWB Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EWB successful.
SGX_PAGE_NOT_BLOCKED	If page is not marked as blocked.
SGX_NOT_TRACKED	If EWB is racing with ETRACK instruction.
SGX_VA_SLOT_OCCUPIED	Version array slot contained valid entry.
SGX_CHILD_PRESENT	Child page present while attempting to page out enclave.

Concurrency Restrictions

Table 40-52. Base Concurrency Restrictions of EWB

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EWB	Source [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	

Table 40-53. Additional Concurrency Restrictions of EWB

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EWB	Source [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Exclusive	

Operation

Temp Variables in EWB Operational Flow

Name	Type	Size (Bytes)	Description
TMP_SRCPGE	Memory page	4096	
TMP_PCMD	PCMD	128	
TMP_SECS	SECS	4096	
TMP_BPEPOCH	UINT64	8	
TMP_BPREFCOUNT	UINT64	8	
TMP_HEADER	MAC Header	128	
TMP_PCMD_ENCLAVEID	UINT64	8	
TMP_VER	UINT64	8	
TMP_PK	UINT128	16	

IF ((DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned))
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
 THEN #PF(DS:RCX); FI;

IF (DS:RDX is not 8Byte Aligned)
 THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
 THEN #PF(DS:RDX); FI;

(* EPCPAGE and VASLOT should not resolve to the same EPC page*)
 IF (DS:RCX and DS:RDX resolve to the same EPC page)
 THEN #GP(0); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
 (* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO *)
 TMP_PCMD := DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)
 THEN #GP(0); FI;

IF ((DS:TMP_PCMD is not 128Byte Aligned) or (DS:TMP_SRCPGE is not 4KByte Aligned))
 THEN #GP(0); FI;

(* Check for concurrent Intel SGX instruction access to the page *)
 IF (Other Intel SGX instruction is accessing page)
 THEN
 IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
 THEN
 VMCS.Exit_reason := SGX_CONFLICT;
 VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
 VMCS.Exit_qualification.error := 0;
 VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;

```

        VMCS.Guest-linear_address := DS:RCX;
    Deliver VMEXIT;
    ELSE
        #GP(0);
    FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        TMP_SECS = Obtain SECS through EPCM(DS:RCX)
        (* Check that EBLOCK has occurred correctly *)
        IF (EBLOCK is not correct)
            THEN #GP(0); FI;
FI;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER) - 1 : 0 ] := 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        (* check to see if the page is evictable *)
        IF (EPCM(DS:RCX).BLOCKED = 0)
            THEN
                RAX := SGX_PAGE NOT_BLOCKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
        FI;
        (* Check if tracking done correctly *)
        IF (Tracking not correct)
            THEN
                RAX := SGX_NOT_TRACKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
        FI;

        (* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *)

```

```

    TMP_HEADER.EID := TMP_SECS.EID;

    (* Obtain EID as an enclave handle for software *)
    TMP_PCMD_ENCLAVEID := TMP_SECS.EID;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
    (*check that there are no child pages inside the enclave *)
    IF (DS:RCX has an EPC page associated with it)
        THEN
            RAX := SGX_CHILD_PRESENT;
            RFLAGS.ZF := 1;
            GOTO ERROR_EXIT;
    FI;
    (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
    IF (<<in VMX non-root operation>> AND
    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
    (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
        THEN
            RFLAGS.ZF := 1;
            RAX := SGX_CHILD_PRESENT;
            GOTO ERROR_EXIT;
    FI;
    TMP_HEADER.EID := 0;
    (* Obtain EID as an enclave handle for software *)
    TMP_PCMD_ENCLAVEID := (DS:RCX).EID;
ELSE IF (EPCM(DS:RCX).PT is PT_VA)
    TMP_HEADER.EID := 0; // Zero is not a special value
    (* No enclave handle for VA pages*)
    TMP_PCMD_ENCLAVEID := 0;
FI;

TMP_HEADER.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE*)
{DS:TMP_SRCPGE, DS:TMP_PCMD.MAC} := AES_GCM_ENC(CR_BASE_PK), (TMP_VER << 32),
    TMP_HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCMD.SECINFO
DS:TMP_PCMD.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
DS:TMP_PCMD.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
DS:TMP_PCMD.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
DS:TMP_PCMD.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
DS:TMP_PCMD.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;
DS:TMP_PCMD.RESERVED := 0;
DS:TMP_PCMD.ENCLAVEID := TMP_PCMD_ENCLAVEID;
DS:RAX.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)

```

```

IF ([DS.RDX])
  THEN
    RAX := SGX_VA_SLOT_OCCUPIED
    RFLAGS.CF := 1;
FI;

```

```

(* Write version to Version Array slot *)
[DS.RDX] := TMP_VER;

```

```

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID := 0;
ERROR_EXIT:

```

Flags Affected

ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.

CF is set if VA slot is previously occupied, Otherwise cleared.

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If the EPC page and VASLOT resolve to the same EPC page. If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages. If the tracking resource is in use. If the EPC page or the version array page is invalid. If the parameters fail consistency checks.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If one of the EPC memory operands has incorrect page type.

64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If a memory operand is non-canonical form. If a memory operand is not properly aligned. If the EPC page and VASLOT resolve to the same EPC page. If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages. If the tracking resource is in use. If the EPC page or the version array page in invalid. If the parameters fail consistency checks.
#PF(error code)	<ul style="list-style-type: none"> If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If one of the EPC memory operands has incorrect page type.

40.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPT Faulting Conditions

The operands are not properly aligned.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	Page type is PT_REG and MODIFIED bit is 0.
SECINFO contains an invalid request.	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
If security attributes of the SECINFO page make the page inaccessible.	

The error codes are:

Table 40-54. EACCEPT Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EACCEPT successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page.

Concurrency Restrictions

Table 40-55. Base Concurrency Restrictions of EACCEPT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target [DS:RCX]	Shared	#GP	
	SECINFO [DS:RBX]	Concurrent		

Table 40-56. Additional Concurrency Restrictions of EACCEPT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)
 THEN #GP(0); FI;

IF (DS:RBX is not within CR_ELRANGE)
 THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
 THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
 (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or
 (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
 (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)))
 THEN #PF(DS:RBX); FI;

(* Copy 64 bytes of contents *)
 SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
 IF (SCRATCH_SECINFO reserved fields are not zero)
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
 THEN #GP(0); FI;

```
IF (DS:RCX is not within CR_ELRANGE)
    THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

(* Check that the combination of requested PT, PENDING and MODIFIED is legal *)

```
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 0 )
    THEN
        IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and
            ((SCRATCH_SECINFO.FLAGS.PR is 1) or
            (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and
            (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or
            ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and
            (SCRATCH_SECINFO.FLAGS.PR is 0) and
            (SCRATCH_SECINFO.FLAGS.PENDING is 0) and
            (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) )))
            THEN #GP(0); FI
```

```
ELSE
```

```
    IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) AND
        ((SCRATCH_SECINFO.FLAGS.PR is 1) OR
        (SCRATCH_SECINFO.FLAGS.PENDING is 1)) AND
        (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) OR
        ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS OR PT_TRIM) AND
        (SCRATCH_SECINFO.FLAGS.PENDING is 0) AND
        (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) AND
        (SCRATCH_SECINFO.FLAGS.PR is 0)) OR
        ((SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST or PT_SS_REST) AND
        (SCRATCH_SECINFO.FLAGS.PENDING is 1) AND
        (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) AND
        (SCRATCH_SECINFO.FLAGS.PR is 0))))
        THEN #GP(0); FI;
```

```
FI;
```

(* Check security attributes of the destination EPC page *)

```
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).BLOCKED is not 0) or
    ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM)
    and (EPCM(DS:RCX).PT is not PT_SS_FIRST) and (EPCM(DS:RCX).PT is not PT_SS_REST)) or
    (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
    THEN #PF(DS:RCX); FI;
```

(* Check the destination EPC page for concurrency *)

```
IF ( EPC page in use )
    THEN #GP(0); FI;
```

(* Re-Check security attributes of the destination EPC page *)

```
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;
```

(* Verify that accept request matches current EPC page settings *)

```
IF ( (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or
    (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or
    (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
```

SGX INSTRUCTION REFERENCES

```
THEN
    RFLAGS.ZF := 1;
    RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
    GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_NOT_TRACKED;
        GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
FI;

(* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
(* check that TCS.PREVSSP is 0 *)
IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0) or
((CPUID.(EAX=12H, ECX=1):EAX[6] = 1) AND ((DS:RCX).PREVSSP != 0)))
    THEN #GP(0); FI;

(* Check consistency of FS & GS Limit *)
IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX).FSLIMIT & FFFH ≠ FFFH) or (DS:RCX).GSLIMIT & FFFH ≠ FFFH) )
    THEN #GP(0); FI;

(* Clear PENDING/MODIFIED flags to mark accept operation complete *)
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0) If executed outside an enclave.
If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.

EACCEPTCOPY—Initialize a Pending Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLU[EACCEPTCOPY]	IR	V/V	SGX2	This leaf function initializes a dynamically allocated EPC page from another page in the EPC.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EACCEPTCOPY (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)	Address of the source EPC page (In)

Description

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

EACCEPTCOPY Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)	EPCPAGE (Source)
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPTCOPY Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	If security attributes of the source EPC page make the page inaccessible.
The EPC page is not valid.	RBX does not contain an effective address in an EPC page in the running enclave.
SECINFO contains an invalid request.	RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

Table 40-57. EACCEPTCOPY Return Value in RAX

Error Code (see Table 40-4)	Description
No Error	EACCEPTCOPY successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.

Concurrency Restrictions

Table 40-58. Base Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPTCOPY	Target [DS:RCX]	Concurrent		
	Source [DS:RDX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Table 40-59. Additional Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPTCOPY	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPTCOPY Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)
THEN #GP(0); FI;

IF ((DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned))
THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) or (DS:RDX is not within CR_ELRANGE))
THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

IF (DS:RDX does not resolve within an EPC)
THEN #PF(DS:RDX); FI;

IF ((EPCM(DS:RBX & ~FFFH).VALID = 0) or (EPCM(DS:RBX & ~FFFH).R = 0) or (EPCM(DS:RBX & ~FFFH).PENDING ≠ 0) or
(EPCM(DS:RBX & ~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX & ~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX & ~FFFH).PT ≠ PT_REG) or
(EPCM(DS:RBX & ~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
(EPCM(DS:RBX & ~FFFH).ENCLAVEADDRESS ≠ DS:RBX))
THEN #PF(DS:RBX); FI;

SGX INSTRUCTION REFERENCES

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ((SCRATCH_SECINFO reserved fields are not zero) or (SCRATCH_SECINFO.FLAGS.R=0) AND(SCRATCH_SECINFO.FLAGS.W#0) or
(SCRATCH_SECINFO.FLAGS.PT is not PT_REG))
THEN #GP(0); FI;

(* Check security attributes of the source EPC page *)
IF ((EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING # 0) or (EPCM(DS:RDX).MODIFIED # 0) or
(EPCM(DS:RDX).BLOCKED # 0) or (EPCM(DS:RDX).PT # PT_REG) or (EPCM(DS:RDX).ENCLAVESECS # CR_ACTIVE_SECS) or
(EPCM(DS:RDX).ENCLAVEADDRESS # DS:RDX))
THEN #PF(DS:RDX); FI;

(* Check security attributes of the destination EPC page *)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
(EPCM(DS:RCX).BLOCKED # 0) or (EPCM(DS:RCX).PT # PT_REG) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS))
THEN
RFLAGS.ZF := 1;
RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
GOTO DONE;
FI;

(* Check the destination EPC page for concurrency *)
IF (destination EPC page in use)
THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
(EPCM(DS:RCX).R # 1) or (EPCM(DS:RCX).W # 1) or (EPCM(DS:RCX).X # 0) or
(EPCM(DS:RCX).PT # SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS) or
(EPCM(DS:RCX).ENCLAVEADDRESS # DS:RCX))
THEN
RFLAGS.ZF := 1;
RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
GOTO DONE;
FI;

(* Copy 4KBbytes form the source to destination EPC page*)
DS:RCX[32767:0] := DS:RDX[32767:0];

(* Update EPCM permissions *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PENDING := 0;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected

Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave. If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If EPC page has incorrect page type or security attributes.

EENTER—Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLU[EENTER]	IR	V/V	SGX1	This leaf function is used to enter an enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	
IR	EENTER (In)	Content of RBX.CSSA (Out)	Address of a TCS (In)	Address of AEP (In)	Address of IP following EENTER (Out)

Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

EENTER Memory Parameter Semantics

TCS
Enclave access

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs. FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 42.1.2):
 - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 42.2.5).
 - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 42.2.2).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):

- All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
- PEBS is suppressed.
- AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set
- If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 40-60. Base Concurrency Restrictions of EENTER

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EENTER	TCS [DS:RBX]	Shared	#GP	

Table 40-61. Additional Concurrency Restrictions of EENTER

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EENTER	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));
```

(* Make sure DS is usable, expand up *)

```
IF (TMP_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
  THEN #GP(0); FI;
```

(* Check that CS, SS, DS, ES.base is 0 *)

```
IF (TMP_MODE64 = 0)
  THEN
    IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
    IF(ES usable and ES.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.B = 0) #GP(0); FI;
```

SGX INSTRUCTION REFERENCES

```
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
```

```

    IF (TMP_GSLIMIT < TMP_GSBASE)
        THEN
            IF (DS.limit < 4GB) THEN #GP(0); FI;
        ELSE
            IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
ELSE
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
        THEN #GP(0); FI;
FI;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFF7FEH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)

```

SGX INSTRUCTION REFERENCES

```

    THEN #PF(DS:TMP_SSA_PAGE); FI;
IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
    THEN #PF(DS:TMP_SSA_PAGE); FI;
IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
    THEN #PF(DS:TMP_SSA_PAGE); FI;
CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

```

(* Compute address of GPR area*)

```
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

If a fault occurs; release locks, abort and deliver that fault;

```

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

```

```

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE - 1) is not in DS segment) THEN #GP(0); FI;
FI;

```

```
CR_GPR_PA := Physical_Address (DS: TMP_GPR);
```

(* Validate TCS.OENTRY *)

```
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
```

```

IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

```

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)

```
IF (DS:RBX.STATE = ACTIVE)
```

```
    THEN #GP(0); FI;
```

```
IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
```

```
    THEN
```

```
        IF ( CR4.CET = 0 )
```

```
            THEN
```

```
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
```

```
                IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI;
```

```
        FI;
```

(* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail EENTER *)

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1)

THEN

IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;

FI;

TMP_IA32_U_CET := 0;

TMP_SSP := 0;

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)

THEN

(* Setup CET state from SECS, note tracker goes to IDLE *)

TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;

IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1)

THEN

TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;

TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;

FI;

(* Compute linear address of what will become new CET state save area and cache its PA *)

TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16

TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

Check the TMP_CET_SAVE_PAGE page is read/write accessible

If fault occurs release locks, abort and deliver fault

(* Read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)

IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR

(EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))

THEN

#PF(DS:TMP_CET_SAVE_PAGE);

FI;

CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)

IF TMP_IA32_U_CET.SH_STK_EN = 1

THEN

TMP_SSP = TCS.PREVSSP;

FI;

FI;

FI;

CR_ENCLAVE_MODE := 1;

CR_ACTIVE_SECS := TMP_SECS;

CR_ELRRANGE := (TMPSECS.BASEADDR, TMP_SECS.SIZE);

(* Save state for possible AEXs *)

SGX INSTRUCTION REFERENCES

```
CR_TCS_PA := Physical_Address (DS:RBX);  
CR_TCS_LA := RBX;  
CR_TCS_LA.AEP := RCX;
```

(* Save the hidden portions of FS and GS *)

```
CR_SAVE_FS_selector := FS.selector;  
CR_SAVE_FS_base := FS.base;  
CR_SAVE_FS_limit := FS.limit;  
CR_SAVE_FS_access_rights := FS.access_rights;  
CR_SAVE_GS_selector := GS.selector;  
CR_SAVE_GS_base := GS.base;  
CR_SAVE_GS_limit := GS.limit;  
CR_SAVE_GS_access_rights := GS.access_rights;
```

(* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM*)

```
IF (CR4.OSXSAVE = 1)  
    CR_SAVE_XCRO := XCRO;  
    XCRO := TMP_SECS.ATTRIBUTES.XFRM;  
FI;
```

```
RCX := RIP;  
RIP := TMP_TARGET;  
RAX := (DS:RBX).CSSA;  
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)  
DS:TMP_SSA.U_RSP := RSP;  
DS:TMP_SSA.U_RBP := RBP;
```

(* Do the FS/GS swap *)

```
FS.base := TMP_FSBASE;  
FS.limit := DS:RBX.FSLIMIT;  
FS.type := 0001b;  
FS.W := DS.W;  
FS.S := 1;  
FS.DPL := DS.DPL;  
FS.G := 1;  
FS.B := 1;  
FS.P := 1;  
FS.AVL := DS.AVL;  
FS.L := DS.L;  
FS.unusable := 0;  
FS.selector := 0BH;
```

```
GS.base := TMP_GSBASE;  
GS.limit := DS:RBX.GSLIMIT;  
GS.type := 0001b;  
GS.W := DS.W;  
GS.S := 1;  
GS.DPL := DS.DPL;  
GS.G := 1;  
GS.B := 1;  
GS.P := 1;  
GS.AVL := DS.AVL;  
GS.L := DS.L;  
GS.unusable := 0;
```

```

GS.selector := OBH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
  THEN
    Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
    CR_SAVE_TF := RFLAGS.TF;
    RFLAGS.TF := 0;
    Suppress_monitor_trap_flag for the source of the execution of the enclave;
    Suppress any pending debug exceptions;
    Suppress any pending MTF VM exit;
  ELSE
    IF RFLAGS.TF = 1
      THEN pend a single-step #DB at the end of EENTER; FI;
    IF the "monitor trap flag" VM-execution control is set
      THEN pend an MTF VM exit at the end of EENTER; FI;
  FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
  THEN
    (* Save enclosing application CET state into save registers *)
    CR_SAVE_IA32_U_CET := IA32_U_CET
    (* Setup enclave CET state *)
    IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
      THEN
        CR_SAVE_SSP := SSP
        SSP := TMP_SSP;
      FI;
    IA32_U_CET := TMP_IA32_U_CET;
  FI;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;

```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> If DS:RBX is not page aligned. If the enclave is not initialized. If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned. If the thread is not in the INACTIVE state. If CS, DS, ES or SS bases are not all zero. If executed in enclave mode. If any reserved field in the TCS FLAG is set. If the target address is not within the CS segment. If CR4.OSFXSR = 0. If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3. If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
--------	--

#PF(error code) If a page fault occurs in accessing memory.
If DS:RBX does not point to a valid TCS.
If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0) If DS:RBX is not page aligned.
If the enclave is not initialized.
If the thread is not in the INACTIVE state.
If CS, DS, ES or SS bases are not all zero.
If executed in enclave mode.
If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
If the target address is not canonical.
If CR4.OSFXSR = 0.
If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

#PF(error code) If a page fault occurs in accessing memory operands.
If DS:RBX does not point to a valid TCS.
If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (Out)

Description

The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode) a #GP(0) results.

EEXIT Memory Parameter Semantics

Target Address
Non-Enclave read and execute access

If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

Concurrency Restrictions

Table 40-62. Base Concurrency Restrictions of EEXIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXIT		Concurrent		

Table 40-63. Additional Concurrency Restrictions of EEXIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXIT		Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

```
IF (TMP_MODE64 = 1)
    THEN
        IF (RBX is not canonical) THEN #GP(0); FI;
    ELSE
        IF (RBX > CS limit) THEN #GP(0); FI;
FI;
```

TMP_RIP := CRIP;
RIP := RBX;

(* Return current AEP in RCX *)
RCX := CR_TCS_PA.AEP;

(* Do the FS/GS swap *)
FS.selector := CR_SAVE_FS.selector;
FS.base := CR_SAVE_FS.base;
FS.limit := CR_SAVE_FS.limit;
FS.access_rights := CR_SAVE_FS.access_rights;
GS.selector := CR_SAVE_GS.selector;
GS.base := CR_SAVE_GS.base;
GS.limit := CR_SAVE_GS.limit;
GS.access_rights := CR_SAVE_GS.access_rights;

(* Restore XCRO if needed *)
IF (CR4.OSXSAVE = 1)
 XCRO := CR_SAVE__XCRO;
FI;

Unsuppress_all_code_breakpoints_that_are_outside_ELRANGE;

```
IF (CR_DBGOPTIN = 0)
    THEN
        UnSuppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        Restore suppressed breakpoint matches;
        RFLAGS.TF := CR_SAVE_TF;
        UnSuppress_montior_trap_flag;
        UnSuppress_LBR_Generation;
        UnSuppress_performance_monitoring_activity;
        Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;
```

```
IF RFLAGS.TF = 1
    THEN Pend Single-Step #DB at the end of EEXIT;
FI;
```

```

IF the "monitor trap flag" VM-execution control is set
  THEN pend a MTF VM exit at the end of EEXIT;
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
  THEN
    (* Record PREVSSP *)
    IF (IA32_U_CET.SH_STK_EN == 1)
      THEN CR_TCS_PA.PREVSSP = SSP; FI;

    (* Restore enclosing apps CET state from the save registers *)
    IA32_U_CET := CR_SAVE_IA32_U_CET;
    IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
      THEN SSP := CR_SAVE_SSP; FI;

    (* Update enclosing apps TRACKER if enclosing app has indirect branch tracking enabled *)
    IF (CR4.CET = 1 AND IA32_U_CET.ENDBR_EN = 1)
      THEN
        IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
        IA32_U_CET.SUPPRESS := 0
      FI;
    FI;

CR_ENCLAVE_MODE := 0;
CR_TCS_PA.STATE := INACTIVE;

(* Assure consistent translations *)
Flush_linear_context;

```

Flags Affected

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If RBX is outside the CS segment.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave. If RBX is not canonical.
#PF(error code)	If a page fault occurs in accessing memory operands.

EGETKEY—Retrieves a Cryptographic Key

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLU[EGETKEY]	IR	V/V	SGX1	This leaf function retrieves a cryptographic key.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EGETKEY (In)	Return error code (Out)	Address to a KEYREQUEST (In)	Address of the OUTPUTDATA (In)

Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

EGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 40-64.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX_INVALID_SVN, SGX_INVALID_ATTRIBUTE, SGX_INVALID_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

Requesting Keys

The KEYREQUEST structure (see Section 37.18.1) identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 40-64) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A “yes” in Table 40-64 indicates the value for the field is included from its default location, identified in the source row, and a “request” indicates the values for the field is included from its corresponding KeyRequest field.

Table 40-64. Key Derivation

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PROPID	ISVEXT PROPID	ISVFAM ILYID	MRENCLAVE	MRSIGNER	CONFIG ID	CONFIGS VN	RAND
Source	Key Dependent Constant	Y := SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;	CR_SGX_OWNER EPOCH	Y := CPUSVN Register;	R := Req.ISV SVN;	SECS.ISVID	SECS.ISVEXTPR ODID	SECS.ISVFAMIL YID	SECS.MRENCLAVE	SECS.MRSIGNER	SECS.CONFIGID	SECS.CONFIGSVN	Req. KEYID
		R := AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;		R := Req.CPU SVN;									
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	No	No	Yes	No	No	Request
Report	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	Request
Seal	Yes	Request	Yes	Request	Request	Request	Request	Request	Request	Request	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	No	No	Yes	No	No	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Request	Request	Request	No	Yes	Request	Request	Yes

Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN_KEY be set and SECS.MRSIGNER equal IA32_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcoded PKCS padding constant (352 byte string):

HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;

HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH

HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;

The error codes are:

Table 40-65. EGETKEY Return Value in RAX

Error Code (see Table 40-4)	Value	Description
No Error	0	EGETKEY successful.
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.
SGX_INVALID_ISVSVN		If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave's corresponding SVN.
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value.

Concurrency Restrictions

Table 40-66. Base Concurrency Restrictions of EGETKEY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		
	OUTPUTDATA [DS:RCX]	Concurrent		

Table 40-67. Additional Concurrency Restrictions of EGETKEY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EGETKEY Operational Flow

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_ISVEXTPRODID		16 bytes	Temp Space for ISVEXTPRODID.
TMP_ISVPRODID		2 bytes	Temp Space for ISVPRODID.
TMP_ISVFAMILYID		16 bytes	Temp Space for ISVFAMILYID.
TMP_CONFIGID		64 bytes	Temp Space for CONFIGID.
TMP_CONFIGSVN		2 bytes	Temp Space for CONFIGSVN.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

(* Make sure KEYREQUEST is properly aligned and inside the current enclave *)

IF ((DS:RBX is not 512Byte aligned) or (DS:RBX is within CR_ELRANGE))
 THEN #GP(0); FI;

(* Make sure DS:RBX is an EPC address and the EPC page is valid *)

IF ((DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0))
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)

THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)

IF ((EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
 (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH)) or (EPCM(DS:RBX).R = 0))
 THEN #PF(DS:RBX);

FI;

(* Make sure OUTPUTDATA is properly aligned and inside the current enclave *)

IF ((DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR_ELRANGE))
 THEN #GP(0); FI;

(* Make sure DS:RCX is an EPC address and the EPC page is valid *)

IF ((DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0))

```

THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
  THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
  (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~OFFFH) ) or (EPCM(DS:RCX).W = 0) )
  THEN #PF(DS:RCX);
FI;

(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
  THEN #GP(0); FI;

TMP_CURRENTSECS := CR_ACTIVE_SECS;

(* Verify that CONFIGSVN & New Policy bits are not used if KSS is not enabled *)
IF ((TMP_CURRENTSECS.ATTRIBUTES.KSS == 0) AND ((DS:RBX.KEYPOLICY & 0x003C ≠ 0) OR (DS:RBX.CONFIGSVN > 0)))
  THEN #GP(0); FI;

(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] := 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES := (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCSELECT fields to be included *)
TMP_MISCSELECT := DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCSELECT

(* Compute CET_ATTRIBUTES fields to be included *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
  THEN TMP_CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES_MASK & TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
TMP_KEYDEPENDENCIES := 0;

CASE (DS:RBX.KEYNAME)
  SEAL_KEY:
    IF (DS:RBX.CPUSVN is beyond current CPU configuration)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
    FI;
    IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;
    IF (DS:RBX.CONFIGSVN > TMP_CURRENTSECS.CONFIGSVN)
      THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;

  (*Include enclave identity?*)

```

SGX INSTRUCTION REFERENCES

```
TMP_MRENCLAVE := 0;
IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
    THEN TMP_MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
FI;
(*Include enclave author?*)
TMP_MRSIGNER := 0;
IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
    THEN TMP_MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
FI;
(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
FI;

(* Include enclave product ID? *)
TMP_ISVPRODID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
    TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
    TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
FI;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1 )
    TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

//Determine values key is based on
TMP_KEYDEPENDENCIES.KEYNAME := SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_MRENCLAVE;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
```

```

TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := DS:RBX.CET_ATTRIBUTES_MASK;
FI;
BREAK;
REPORT_KEY:
//Determine values key is based on
TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
EINITTOKEN_KEY:
(* Check ENCLAVE has EINITTOKEN Key capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.EINITTOKEN_KEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;

```

```

(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
PROVISION_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;
(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;

```

```

TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := 0;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
PROVISION_SEAL_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
FI;
(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
FI;

(* Include enclave product ID? *)
TMP_ISVPRODID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
    THEN TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    THEN TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;

```

SGX INSTRUCTION REFERENCES

```
TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
FI;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
    TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTESMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
    FI;
BREAK;
DEFAULT:
    (* The value of KEYNAME is invalid *)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_KEYNAME;
    GOTO EXIT;
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY := derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] := TMP_OUTPUTKEY;
RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the current enclave. If an effective address is not properly aligned. If an effective address is outside the DS segment limit. If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave. If a memory operand effective address is outside the current enclave. If an effective address is not properly aligned. If an effective address is not canonical. If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory operands.

EMODPE—Extend an EPC Page Permissions

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLU[EMODPE]	IR	V/V	SGX2	This leaf function extends the access rights of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EMODPE (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

EMODPE Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EMODPE Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	

Concurrency Restrictions

Table 40-68. Base Concurrency Restrictions of EMODPE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPE	Target [DS:RCX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Table 40-69. Additional Concurrency Restrictions of EMODPE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPE	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EMODPE Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE))
THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); FI;

IF ((EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).R = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or
(EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
(EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)))
THEN #PF(DS:RBX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero)
THEN #GP(0); FI;

(* Check security attributes of the EPC page *)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
(EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
THEN #GP(0); FI;

(* Re-Check security attributes of the EPC page *)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
(EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
(EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
THEN #PF(DS:RCX); FI;

(* Check for misconfigured SECINFO flags*)
IF ((EPCM(DS:RCX).R = 0) and (SCRATCH_SECINFO.FLAGS.R = 0) and (SCRATCH_SECINFO.FLAGS.W ≠ 0))
THEN #GP(0); FI;

(* Update EPCM permissions *)

EPCM(DS:RCX).R := EPCM(DS:RCX).R | SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := EPCM(DS:RCX).W | SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := EPCM(DS:RCX).X | SCRATCH_SECINFO.FLAGS.X;

Flags Affected

None

Protected Mode Exceptions

- #GP(0) If executed outside an enclave.
If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If a memory operand is locked.
- #PF(error code) If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

- #GP(0) If executed outside an enclave.
If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If a memory operand is locked.
- #PF(error code) If a page fault occurs in accessing memory operands.

EReport—Create a Cryptographic Report of the Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLU[EReport]	IR	V/V	SGX1	This leaf function creates a cryptographic report of the enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX	RDX
IR	EReport (In)	Address of TARGETINFO (In)	Address of REPORTDATA (In)	Address where the REPORT is written to in an OUTPUTDATA (In)

Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

EReport Memory Parameter Semantics

TARGETINFO	REPORTDATA	OUTPUTDATA
Read access by Enclave	Read access by Enclave	Read/Write access by Enclave

This instruction leaf perform the following:

1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a cryptographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.

CR_REPORT_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

The instruction faults if any of the following:

EReport Faulting Conditions

An effective address not properly aligned.	An memory address does not resolve in an EPC page.
If accessing an invalid EPC page.	If the EPC page is blocked.
May page fault.	

Concurrency Restrictions

Table 40-70. Base Concurrency Restrictions of EREPORT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREPORT	TARGETINFO [DS:RBX]	Concurrent		
	REPORTDATA [DS:RCX]	Concurrent		
	OUTPUTDATA [DS:RDX]	Concurrent		

Table 40-71. Additional Concurrency Restrictions of EREPORT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREPORT	TARGETINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RDX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREPORT Operational Flow

Name	Type	Size (bits)	Description
TMP_ATTRIBUTES		32	Physical address of SECS of the enclave to which source operand belongs.
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_REPORTKEY		128	REPORTKEY generated by the instruction.
TMP_REPORT		3712	

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Address verification for TARGETINFO (RBX) *)

IF ((DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR_ELRange))
 THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).VALID = 0)
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
 THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)

IF ((EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
 (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~OFFFH)) or (EPCM(DS:RBX).R = 0))

```

THEN #PF(DS:RBX);
FI;

```

```

(* Verify RESERVED spaces in TARGETINFO are valid *)

```

```

IF (DS:RBX.RESERVED != 0)
  THEN #GP(0); FI;

```

```

(* Address verification for REPORTDATA (RCX) *)

```

```

IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
  THEN #GP(0); FI;

```

```

IF (DS:RCX does not resolve within an EPC)

```

```

  THEN #PF(DS:RCX); FI;

```

```

IF (EPCM(DS:RCX).VALID = 0)

```

```

  THEN #PF(DS:RCX); FI;

```

```

IF (EPCM(DS:RCX).BLOCKED = 1)

```

```

  THEN #PF(DS:RCX); FI;

```

```

(* Check page parameters for correctness *)

```

```

IF ( (EPCM(DS:RCX).PT != PT_REG) or (EPCM(DS:RCX).ENCLAVESECS != CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
  (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS != (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).R = 0) )
  THEN #PF(DS:RCX);

```

```

FI;

```

```

(* Address verification for OUTPUTDATA (RDX) *)

```

```

IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
  THEN #GP(0); FI;

```

```

IF (DS:RDX does not resolve within an EPC)

```

```

  THEN #PF(DS:RDX); FI;

```

```

IF (EPCM(DS:RDX).VALID = 0)

```

```

  THEN #PF(DS:RDX); FI;

```

```

IF (EPCM(DS:RDX).BLOCKED = 1)

```

```

  THEN #PF(DS:RDX); FI;

```

```

(* Check page parameters for correctness *)

```

```

IF ( (EPCM(DS:RDX).PT != PT_REG) or (EPCM(DS:RDX).ENCLAVESECS != CR_ACTIVE_SECS) or (EPCM(DS:RDX).PENDING = 1) or
  (EPCM(DS:RDX).MODIFIED = 1) or (EPCM(DS:RDX).ENCLAVEADDRESS != (DS:RDX & ~0FFFH) ) or (EPCM(DS:RDX).W = 0) )
  THEN #PF(DS:RDX);

```

```

FI;

```

```

(* REPORT MAC needs to be computed over data which cannot be modified *)

```

```

TMP_REPORT.CPUSVN := CR_CPUSVN;

```

```

TMP_REPORT.ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;

```

```

TMP_REPORT.ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;

```

```

TMP_REPORT.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;

```

```

TMP_REPORT.ISVSVN := TMP_CURRENTSECS.ISVSVN;

```

```

TMP_REPORT.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;

```

```

TMP_REPORT.REPORTDATA := DS:RCX[511:0];

```

```

TMP_REPORT.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;

```

SGX INSTRUCTION REFERENCES

```
TMP_REPORT.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED := 0;
TMP_REPORT.KEYID[255:0] := CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_REPORT.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_REPORT.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_REPORT.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
```

(* Derive the report key *)

```
TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNERPOUCH := CR_SGXOWNERPOUCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := DS:RBX.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := DS:RBX.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
```

(* Calculate the derived key*)

```
TMP_REPORTKEY := derivekey(TMP_KEYDEPENDENCIES);
```

(* call cryptographic CMAC function, CMAC data are not including MAC&KEYID *)

```
TMP_REPORT.MAC := cmac(TMP_REPORTKEY, TMP_REPORT[3071:0]);
DS:RDX[3455:0] := TMP_REPORT;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave. If the address in RCS is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave. If RCX is non-canonical form. If a memory operand is not properly aligned. If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

ERESUME—Re-Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLU[ERESUME]	IR	V/V	SGX1	This leaf function is used to re-enter an enclave after an interrupt.

Instruction Operand Encoding

Op/En	RAX	RBX	RCX
IR	ERESUME (In)	Address of a TCS (In)	Address of AEP (In)

Description

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

ERESUME Memory Parameter Semantics

TCS
Enclave read/write access

The instruction faults if any of the following:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs. FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 42.1.2):
 - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 42.2.5).
 - On opt-in entry, a single-step debug exception is pending on the instruction boundary immediately after EENTER (see Section 42.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.

- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):
 - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
 - PEBS is suppressed.
 - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set.
 - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 40-72. Base Concurrency Restrictions of ERESUME

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERESUME	TCS [DS:RBX]	Shared	#GP	

Table 40-73. Additional Concurrency Restrictions of ERESUME

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERESUME	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)

IF (TMP_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
 THEN #GP(0); FI;

SGX INSTRUCTION REFERENCES

(* Check that CS, SS, DS, ES.base is 0 *)

```
IF (TMP_MODE64 = 0)
  THEN
    IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
    IF(ES usable and ES.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.B = 0) #GP(0); FI;
FI;
```

IF (DS:RBX is not 4KByte Aligned)

```
  THEN #GP(0); FI;
```

IF (DS:RBX does not resolve within an EPC)

```
  THEN #PF(DS:RBX); FI;
```

(* Check AEP is canonical*)

```
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
  THEN #GP(0); FI;
```

(* Check concurrency of TCS operation*)

```
IF (Other Intel SGX instructions is operating on TCS)
  THEN #GP(0); FI;
```

(* TCS verification *)

```
IF (EPCM(DS:RBX).VALID = 0)
  THEN #PF(DS:RBX); FI;
```

IF (EPCM(DS:RBX).BLOCKED = 1)

```
  THEN #PF(DS:RBX); FI;
```

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))

```
  THEN #PF(DS:RBX); FI;
```

IF ((EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS))

```
  THEN #PF(DS:RBX); FI;
```

IF ((DS:RBX).OSSA is not 4KByte Aligned)

```
  THEN #GP(0); FI;
```

(* Check proposed FS and GS *)

```
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
  THEN #GP(0); FI;
```

(* Get the SECS for the enclave in which the TCS resides *)

```
TMP_SECS := Address of SECS for TCS;
```

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)

```
IF ( ( (DS:RBX).FLAGS & FFFFFFFF00000000H) ≠ 0)
  THEN #GP(0); FI;
```

(* SECS must exist and enclave must have previously been EINITted *)

```
IF (the enclave is not already initialized)
  THEN #GP(0); FI;
```

(* make sure the logical processor's operating mode matches the enclave *)

```
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
  THEN #GP(0); FI;
```

```
IF (CR4.OSFXSR = 0)
  THEN #GP(0); FI;
```

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)

```
IF (CR4.OSXSAVE = 0)
  THEN
    IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
  ELSE
    IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCRO) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
  FI;
```

(* Make sure the SSA contains at least one active frame *)

```
IF ( (DS:RBX).CSSA = 0)
  THEN #GP(0); FI;
```

(* Compute linear address of SSA frame *)

```
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);
```

```
FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
```

```
  (* Check page is read/write accessible *)
```

```
  Check that DS:TMP_SSA_PAGE is read/write accessible;
```

```
  If a fault occurs, release locks, abort and deliver that fault;
```

```
  IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
  CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
```

```
ENDFOR
```

(* Compute address of GPR area*)

```
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

```
Check that DS:TMP_SSA_PAGE is read/write accessible;
```

```
If a fault occurs, release locks, abort and deliver that fault;
```

```
IF (DS:TMP_GPR does not resolve to EPC page)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).VALID = 0)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

```
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
```

```
  THEN #PF(DS:TMP_GPR); FI;
```

SGX INSTRUCTION REFERENCES

```
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
(EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
(EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
THEN #PF(DS:TMP_GPR); FI;
```

```
IF (TMP_MODE64 = 0)
THEN
    IF (TMP_GPR + (GPR_SIZE - 1) is not in DS segment) THEN #GP(0); FI;
FI;
```

```
CR_GPR_PA := Physical_Address (DS: TMP_GPR);
```

```
TMP_TARGET := (DS:TMP_GPR).RIP;
IF (TMP_MODE64 = 1)
THEN
    IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
ELSE
    IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;
```

(* Check proposed FS/GS segments fall within DS *)

```
IF (TMP_MODE64 = 0)
THEN
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
    (* if FS wrap-around, make sure DS has no holes*)
    IF (TMP_FSLIMIT < TMP_FSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
    (* if GS wrap-around, make sure DS has no holes*)
    IF (TMP_GSLIMIT < TMP_GSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
ELSE
    TMP_FSBASE := DS:TMP_GPR.FSBASE;
    TMP_GSBASE := DS:TMP_GPR.GSBASE;
    IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
    THEN #GP(0); FI;
FI;
```

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)

```
IF (DS:RBX.STATE = ACTIVE)
THEN #GP(0); FI;
```

```
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
THEN
```

```

IF ( CR4.CET = 0 )
  THEN
    (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
    IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI;
  FI;
(* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 )
  THEN
    IF ( DS:RBX.OCETSSA is not 16B aligned ) #GP(0); FI;
  FI;

TMP_IA32_U_CET := 0;
TMP_SSP := 0;

```

```

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
  THEN

```

```

  (* Setup CET state from SECS, note tracker goes to IDLE *)
  TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
  IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 )
    THEN
      TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
      TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
    FI;

```

```

  (* Compute linear address of what will become new CET state save area and cache its PA *)
  TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16
  TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

```

Check the TMP_CET_SAVE_PAGE page is read/write accessible
 If fault occurs release locks, abort and deliver fault

```

(* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
  (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
  THEN
    #PF(DS:TMP_CET_SAVE_PAGE);
  FI;

```

```

CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)

```

```

TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;

```

```

IF ( (TMP_MODE64 = 1 AND TMP_SSP is not canonical) OR
  (TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFFF0000000) ≠ 0) OR

```

SGX INSTRUCTION REFERENCES

```
(TMP_SSP is not 4 byte aligned) OR  
(TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR  
(CR_CET_SAVE_AREA_PA.Reserved ≠ 0) ) #GP(0); FI;  
FI;
```

```
(* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)  
(* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)  
XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);
```

```
IF (XRSTOR failed with #GP)  
  THEN  
    DS:RBX.STATE := INACTIVE;  
    #GP(0);
```

```
FI;
```

```
CR_ENCLAVE_MODE := 1;  
CR_ACTIVE_SECS := TMP_SECS;  
CR_EL RANGE := (TMP_SECS.BASEADDR, TMP_SECS.SIZE);
```

```
(* Save state for possible AEXs *)  
CR_TCS_PA := Physical_Address (DS:RBX);  
CR_TCS_LA := RBX;  
CR_TCS_LA.AEP := RCX;
```

```
(* Save the hidden portions of FS and GS *)  
CR_SAVE_FS_selector := FS.selector;  
CR_SAVE_FS_base := FS.base;  
CR_SAVE_FS_limit := FS.limit;  
CR_SAVE_FS_access_rights := FS.access_rights;  
CR_SAVE_GS_selector := GS.selector;  
CR_SAVE_GS_base := GS.base;  
CR_SAVE_GS_limit := GS.limit;  
CR_SAVE_GS_access_rights := GS.access_rights;
```

```
RIP := TMP_TARGET;
```

```
Restore_GPRs from DS:TMP_GPR;
```

```
(*Restore the RFLAGS values from SSA*)  
RFLAGS.CF := DS:TMP_GPR.RFLAGS.CF;  
RFLAGS.PF := DS:TMP_GPR.RFLAGS.PF;  
RFLAGS.AF := DS:TMP_GPR.RFLAGS.AF;  
RFLAGS.ZF := DS:TMP_GPR.RFLAGS.ZF;  
RFLAGS.SF := DS:TMP_GPR.RFLAGS.SF;  
RFLAGS.DF := DS:TMP_GPR.RFLAGS.DF;  
RFLAGS.OF := DS:TMP_GPR.RFLAGS.OF;  
RFLAGS.NT := DS:TMP_GPR.RFLAGS.NT;  
RFLAGS.AC := DS:TMP_GPR.RFLAGS.AC;  
RFLAGS.ID := DS:TMP_GPR.RFLAGS.ID;  
RFLAGS.RF := DS:TMP_GPR.RFLAGS.RF;  
RFLAGS.VM := 0;  
IF (RFLAGS.IOPL = 3)  
  THEN RFLAGS.IF := DS:TMP_GPR.RFLAGS.IF; FI;
```

```

IF (TCS.FLAGS.OPTIN = 0)
    THEN RFLAGS.TF := 0; FI;

(* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCRO := XCRO;
    XCRO := TMP_SECS.ATTRIBUTES.XFRM;
FI;

(* Pop the SSA stack*)
(DS:RBX).CSSA := (DS:RBX).CSSA - 1;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE

```


SGX INSTRUCTION REFERENCES

```
    Clear all pending debug exceptions;
    Clear pending MTF VM exits;
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP;
            FI;
        IA32_U_CET := TMP_IA32_U_CET;
    FI;
```

```
(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned. If the enclave is not initialized. If the thread is not in the INACTIVE state. If CS, DS, ES or SS bases are not all zero. If executed in enclave mode. If part or all of the FS or GS segment specified by TCS is outside the DS segment. If any reserved field in the TCS FLAG is set. If the target address is not within the CS segment. If CR4.OSFXSR = 0. If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3. If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory. If DS:RBX does not point to a valid TCS. If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned. If the enclave is not initialized. If the thread is not in the INACTIVE state. If CS, DS, ES or SS bases are not all zero. If executed in enclave mode. If part or all of the FS or GS segment specified by TCS is outside the DS segment. If any reserved field in the TCS FLAG is set.
--------	---

If the target address is not canonical.

If CR4.OSFXSR = 0.

If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.

If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

#PF(error code)

If a page fault occurs in accessing memory operands.

If DS:RBX does not point to a valid TCS.

If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

40.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLV instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLV[EDECVIRTCHILD]	IR	V/V	EAX[5]	This leaf function decrements the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDECVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction decrements the SECS VIRTCHILDCNT field. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

EDECVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EDECVIRTCHILD Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions**Table 40-74. Base Concurrency Restrictions of EDECVIRTCHILD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RCX]	Concurrent		

Table 40-75. Additional Concurrency Restrictions of EDECVIRTCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDECVIRTCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_VIRTCHILDCNT	Integer	64	Number of virtual child pages.

EDECVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EDECVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_INVALID_COUNTER		Attempt to decrement counter that is already zero.

(* check alignment of DS:RBX *)

```
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;
```

(* check DS:RBX is an linear address of an EPC page *)

```
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(* check DS:RCX is an linear address of an EPC page *)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(* Check the EPCPAGE for concurrency *)

```
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(* check that the EPC page is valid *)

```
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)

```
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
```

```

(EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
  THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS := Address_of_SECS_for(DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS := Physical_Address(DS:RBX);
ELSE
    (* EDECVIRTUALCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically decrement virtchild counter and check for underflow *)
Locked_Decrement(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an underflow) THEN
    Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_COUNTER;
    goto DONE;
FI;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

```

Flags Affected

ZF is set if EDECVIRTUALCHILD fails due to concurrent operation with another SGX instruction, or if there is a VIRTCHILDCNT underflow. Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If DS segment is unusable. If a memory operand is not properly aligned. RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands. If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS). If RCX does not refer to an SECS page.

64-Bit Mode Exceptions

- #GP(0) If a memory address is in a non-canonical form.
 If a memory operand is not properly aligned.
 RBX does not refer to an enclave page associated with SECS referenced in RCX.
- #PF(error code) If a page fault occurs in accessing memory operands.
 If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
 If RCX does not refer to an SECS page.

EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLV[EINCVIRTCHILD]	IR	V/V	EAX[5]	This leaf function increments the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EINCVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction increments the SECS VIRTCHILDCNT field. This instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create a linear address. Segment override is not supported.

EINCVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EINCVIRTCHILD Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions

Table 40-76. Base Concurrency Restrictions of EINCVIRTCHILD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINCVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RCX]	Concurrent		

Table 40-77. Additional Concurrency Restrictions of EINCVIRTUALCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINCVIRTUALCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EINCVIRTUALCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

EINCVIRTUALCHILD Return Value in RAX

Error	Value	Description
No Error	0	EINCVIRTUALCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

(* check alignment of DS:RBX *)

```
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;
```

(* check DS:RBX is an linear address of an EPC page *)

```
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(* check DS:RCX is an linear address of an EPC page *)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(* Check the EPCPAGE for concurrency *)

```
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(* check that the EPC page is valid *)

```
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;
```

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)

```
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
```

```

THEN
  (* get the SECS of DS:RBX *)
  TMP_SECS := Address_of_SECS_for_DS:RBX;
ELSE IF (EPCM( DS:RBX ).PAGE_TYPE = PT_SECS) THEN
  (* get the physical address of DS:RBX *)
  TMP_SECS := Physical_Address( DS:RBX );
ELSE
  (* EINCVIRTCHILD called on page of incorrect type *)
  #PF( DS:RBX, PFEC.SGX ); FI;

IF ( TMP_SECS ≠ Physical_Address( DS:RCX ) ) THEN
  #GP( 0 ); FI;

(* Atomically increment virtchild counter *)
Locked_Increment( SECS( TMP_SECS ).VIRTCHILDCNT );

```

```

RFLAGS.ZF := 0;
RAX := 0;

```

```

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

```

Flags Affected

ZF is set if EINCVIRTCHILD fails due to concurrent operation with another SGX instruction; otherwise cleared.

Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If DS segment is unusable.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLV[ESETCONTEXT]	IR	V/V	EAX[5]	This leaf function sets the ENCLAVECONTEXT field in SECS.

Instruction Operand Encoding

Op/En	EAX		RCX	RDX
IR	ESETCONTEXT (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Context Value (In, EA)

Description

The ESETCONTEXT leaf overwrites the ENCLAVECONTEXT field in the SECS. ECREATE and ELD of an SECS set the ENCLAVECONTEXT field in the SECS to the address of the SECS (for access later in ERDINFO). The ESETCONTEXT instruction allows a VMM to overwrite the default context value if necessary, for example, if the VMM is emulating ECREATE or ELD on behalf of the guest.

The content of RCX is an effective address of the SECS page to be updated, RDX contains the address pointing to the value to be stored in the SECS. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if:

- The operand is not properly aligned.
- RCX does not refer to an SECS page.

ESETCONTEXT Memory Parameter Semantics

EPCPAGE	CONTEXT
Read access permitted by Enclave	Read/Write access permitted by Non Enclave

The instruction faults if any of the following:

ESETCONTEXT Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

Concurrency Restrictions

Table 40-78. Base Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ESETCONTEXT	SECS [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Table 40-79. Additional Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ESETCONTEXT	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ESETCONTEXT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_CONTEXT	CONTEXT	64	Data Value of CONTEXT.

ESETCONTEXT Return Value in RAX

Error	Value	Description
No Error	0	ESETCONTEXT Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

(* check alignment of the EPCPAGE (RCX) *)

```
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;
```

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)

```
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;
```

(* check alignment of the CONTEXT field (RDX) *)

```
IF (DS:RDX is not 8Byte Aligned) THEN
    #GP(0); FI;
```

(* Load CONTEXT into local variable *)

```
TMP_CONTEXT := DS:RDX
```

(* Check the EPC page for concurrency *)

```
IF (EPC page is being modified) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;
```

(* check page validity *)

```
IF (EPCM(DS:RCX).VALID = 0) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;
```

(* check EPC page is an SECS page *)

SGX INSTRUCTION REFERENCES

```
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN  
  #PF(DS:RCX, PFEC.SGX);  
FI;
```

```
(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)  
SECS(DS:RCX).ENCLAVECONTEXT := TMP_CONTEXT;
```

```
RAX := 0;  
RFLAGS.ZF := 0;
```

```
DONE:  
(* clear flags *)  
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared.
CF, PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If DS segment is unusable. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

35. Updates to Appendix B, Volume 3D

Change bars show changes to Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D: System Programming Guide, Part 4*.

Changes to this chapter: Added information for ENCLV-exiting bitmap to Table B-4 “Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb)”. Added information for Guest IA32_PKRS to Table B-6 “Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb)”. Added information for Host IA32_PKRS to Table B-7 “Encodings for 64-Bit Host-State Fields (0010_11xx_xxxx_xxxAb)”.

Every component of the VMCS is encoded by a 32-bit field that can be used by VMREAD and VMWRITE. Section 24.11.2 describes the structure of the encoding space (the meanings of the bits in each 32-bit encoding).

This appendix 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.)

B.1 16-BIT FIELDS

A value of 0 in bits 14:13 of an encoding indicates a 16-bit field. Only guest-state areas and the host-state area contain 16-bit fields. As noted in Section 24.11.2, each 16-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.1.1 16-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-1 enumerates the 16-bit control fields.

Table B-1. Encoding for 16-Bit Control Fields (0000_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
Virtual-processor identifier (VPID) ¹	000000000B	00000000H
Posted-interrupt notification vector ²	000000001B	00000002H
EPTP index ³	000000010B	00000004H

NOTES:

1. This field exists only on processors that support the 1-setting of the “enable VPID” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.

B.1.2 16-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-2 enumerates 16-bit guest-state fields.

Table B-2. Encodings for 16-Bit Guest-State Fields (0000_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest ES selector	000000000B	00000800H
Guest CS selector	000000001B	00000802H
Guest SS selector	000000010B	00000804H
Guest DS selector	000000011B	00000806H
Guest FS selector	000000100B	00000808H
Guest GS selector	000000101B	0000080AH
Guest LDTR selector	000000110B	0000080CH
Guest TR selector	000000111B	0000080EH

Table B-2. Encodings for 16-Bit Guest-State Fields (0000_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest interrupt status ¹	000001000B	00000810H
PML index ²	000001001B	00000812H

NOTES:

1. This field exists only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “enable PML” VM-execution control.

B.1.3 16-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-3 enumerates the 16-bit host-state fields.

Table B-3. Encodings for 16-Bit Host-State Fields (0000_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host ES selector	000000000B	00000C00H
Host CS selector	000000001B	00000C02H
Host SS selector	000000010B	00000C04H
Host DS selector	000000011B	00000C06H
Host FS selector	000000100B	00000C08H
Host GS selector	000000101B	00000C0AH
Host TR selector	000000110B	00000C0CH

B.2 64-BIT FIELDS

A value of 1 in bits 14:13 of an encoding indicates a 64-bit field. There are 64-bit fields only for controls and for guest state. As noted in Section 24.11.2, every 64-bit field has two encodings, which differ on bit 0, the access type. Thus, each such field has an even encoding for full access and an odd encoding for high access.

B.2.1 64-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-4 enumerates the 64-bit control fields.

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb)

Field Name	Index	Encoding
Address of I/O bitmap A (full)	000000000B	00002000H
Address of I/O bitmap A (high)		00002001H
Address of I/O bitmap B (full)	000000001B	00002002H
Address of I/O bitmap B (high)		00002003H
Address of MSR bitmaps (full) ¹	000000010B	00002004H
Address of MSR bitmaps (high) ¹		00002005H
VM-exit MSR-store address (full)	000000011B	00002006H
VM-exit MSR-store address (high)		00002007H

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) (Contd.)

Field Name	Index	Encoding
VM-exit MSR-load address (full)	000000100B	00002008H
VM-exit MSR-load address (high)		00002009H
VM-entry MSR-load address (full)	000000101B	0000200AH
VM-entry MSR-load address (high)		0000200BH
Executive-VMCS pointer (full)	000000110B	0000200CH
Executive-VMCS pointer (high)		0000200DH
PML address (full) ²	000000111B	0000200EH
PML address (high) ²		0000200FH
TSC offset (full)	000001000B	00002010H
TSC offset (high)		00002011H
Virtual-APIC address (full) ³	000001001B	00002012H
Virtual-APIC address (high) ³		00002013H
APIC-access address (full) ⁴	000001010B	00002014H
APIC-access address (high) ⁴		00002015H
Posted-interrupt descriptor address (full) ⁵	000001011B	00002016H
Posted-interrupt descriptor address (high) ⁵		00002017H
VM-function controls (full) ⁶	000001100B	00002018H
VM-function controls (high) ⁶		00002019H
EPT pointer (EPTP; full) ⁷	000001101B	0000201AH
EPT pointer (EPTP; high) ⁷		0000201BH
EOI-exit bitmap 0 (EOI_EXIT0; full) ⁸	000001110B	0000201CH
EOI-exit bitmap 0 (EOI_EXIT0; high) ⁸		0000201DH
EOI-exit bitmap 1 (EOI_EXIT1; full) ⁸	000001111B	0000201EH
EOI-exit bitmap 1 (EOI_EXIT1; high) ⁸		0000201FH
EOI-exit bitmap 2 (EOI_EXIT2; full) ⁸	000010000B	00002020H
EOI-exit bitmap 2 (EOI_EXIT2; high) ⁸		00002021H
EOI-exit bitmap 3 (EOI_EXIT3; full) ⁸	000010001B	00002022H
EOI-exit bitmap 3 (EOI_EXIT3; high) ⁸		00002023H
EPTP-list address (full) ⁹	000010010B	00002024H
EPTP-list address (high) ⁹		00002025H
VMREAD-bitmap address (full) ¹⁰	000010011B	00002026H
VMREAD-bitmap address (high) ¹⁰		00002027H
VMWRITE-bitmap address (full) ¹⁰	000010100B	00002028H
VMWRITE-bitmap address (high) ¹⁰		00002029H
Virtualization-exception information address (full) ¹¹	000010101B	0000202AH
Virtualization-exception information address (high) ¹¹		0000202BH
XSS-exiting bitmap (full) ¹²	000010110B	0000202CH
XSS-exiting bitmap (high) ¹²		0000202DH

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) (Contd.)

Field Name	Index	Encoding
ENCLS-exiting bitmap (full) ¹³	000010111B	0000202EH
ENCLS-exiting bitmap (high) ¹³		0000202FH
Sub-page-permission-table pointer (full) ¹⁴	000011000B	00002030H
Sub-page-permission-table pointer (high) ¹⁴		00002031H
TSC multiplier (full) ¹⁵	000011001B	00002032H
TSC multiplier (high) ¹⁵		00002033H
ENCLV-exiting bitmap (full) ¹⁶	000011011B	00002036H
ENCLV-exiting bitmap (high) ¹⁶		00002037H

NOTES:

1. This field exists only on processors that support the 1-setting of the “use MSR bitmaps” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “enable PML” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “virtualize APIC accesses” VM-execution control.
5. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
6. This field exists only on processors that support the 1-setting of the “enable VM functions” VM-execution control.
7. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.
8. This field exists only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control.
9. This field exists only on processors that support the 1-setting of the “EPTP switching” VM-function control.
10. This field exists only on processors that support the 1-setting of the “VMCS shadowing” VM-execution control.
11. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
12. This field exists only on processors that support the 1-setting of the “enable XSAVES/XRSTORS” VM-execution control.
13. This field exists only on processors that support the 1-setting of the “enable ENCLS exiting” VM-execution control.
14. This field exists only on processors that support the 1-setting of the “sub-page write permissions for EPT” VM-execution control.
15. This field exists only on processors that support the 1-setting of the “use TSC scaling” VM-execution control.
16. This field exists only on processors that support the 1-setting of the “enable ENCLV exiting” VM-execution control.

B.2.2 64-Bit Read-Only Data Field

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. There is only one such 64-bit field as given in Table B-5. (As with other 64-bit fields, this one has two encodings.)

Table B-5. Encodings for 64-Bit Read-Only Data Field (0010_01xx_xxxx_xxxAb)

Field Name	Index	Encoding
Guest-physical address (full) ¹	000000000B	00002400H
Guest-physical address (high) ¹		00002401H

NOTES:

1. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.

B.2.3 64-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-6 enumerates the 64-bit guest-state fields.

Table B-6. Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb)

Field Name	Index	Encoding
VMCS link pointer (full)	000000000B	00002800H
VMCS link pointer (high)		00002801H
Guest IA32_DEBUGCTL (full)	000000001B	00002802H
Guest IA32_DEBUGCTL (high)		00002803H
Guest IA32_PAT (full) ¹	000000010B	00002804H
Guest IA32_PAT (high) ¹		00002805H
Guest IA32_EFER (full) ²	000000011B	00002806H
Guest IA32_EFER (high) ²		00002807H
Guest IA32_PERF_GLOBAL_CTRL (full) ³	000000100B	00002808H
Guest IA32_PERF_GLOBAL_CTRL (high) ³		00002809H
Guest PDPTE0 (full) ⁴	000000101B	0000280AH
Guest PDPTE0 (high) ⁴		0000280BH
Guest PDPTE1 (full) ⁴	000000110B	0000280CH
Guest PDPTE1 (high) ⁴		0000280DH
Guest PDPTE2 (full) ⁴	000000111B	0000280EH
Guest PDPTE2 (high) ⁴		0000280FH
Guest PDPTE3 (full) ⁴	000001000B	00002810H
Guest PDPTE3 (high) ⁴		00002811H
Guest IA32_BNDCFGS (full) ⁵	000001001B	00002812H
Guest IA32_BNDCFGS (high) ⁵		00002813H
Guest IA32_RTIT_CTL (full) ⁶	000001010B	00002814H
Guest IA32_RTIT_CTL (high) ⁶		00002815H
Guest IA32_PKRS (full) ⁷	000001100B	00002818H
Guest IA32_PKRS (high) ⁷		00002819H

NOTES:

1. This field exists only on processors that support either the 1-setting of the "load IA32_PAT" VM-entry control or that of the "save IA32_PAT" VM-exit control.
2. This field exists only on processors that support either the 1-setting of the "load IA32_EFER" VM-entry control or that of the "save IA32_EFER" VM-exit control.
3. This field exists only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-entry control.
4. This field exists only on processors that support the 1-setting of the "enable EPT" VM-execution control.
5. This field exists only on processors that support either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control.
6. This field exists only on processors that support either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control.
7. This field exists only on processors that support the 1-setting of the "load PKRS" VM-entry control.

B.2.4 64-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-7 enumerates the 64-bit control fields.

Table B-7. Encodings for 64-Bit Host-State Fields (0010_11xx_xxxx_xxxAb)

Field Name	Index	Encoding
Host IA32_PAT (full) ¹	000000000B	00002C00H
Host IA32_PAT (high) ¹		00002C01H
Host IA32_EFER (full) ²	000000001B	00002C02H
Host IA32_EFER (high) ²		00002C03H
Host IA32_PERF_GLOBAL_CTRL (full) ³	000000010B	00002C04H
Host IA32_PERF_GLOBAL_CTRL (high) ³		00002C05H
Host IA32_PKRS (full) ⁴	000000011B	00002C06H
Host IA32_PKRS (high) ⁴		00002C07H

NOTES:

1. This field exists only on processors that support the 1-setting of the "load IA32_PAT" VM-exit control.
2. This field exists only on processors that support the 1-setting of the "load IA32_EFER" VM-exit control.
3. This field exists only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-exit control.
4. This field exists only on processors that support the 1-setting of the "load PKRS" VM-exit control.

B.3 32-BIT FIELDS

A value of 2 in bits 14:13 of an encoding indicates a 32-bit field. As noted in Section 24.11.2, each 32-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.3.1 32-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-8 enumerates the 32-bit control fields.

Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
Pin-based VM-execution controls	000000000B	00004000H
Primary processor-based VM-execution controls	000000001B	00004002H
Exception bitmap	000000010B	00004004H
Page-fault error-code mask	000000011B	00004006H
Page-fault error-code match	000000100B	00004008H
CR3-target count	000000101B	0000400AH
VM-exit controls	000000110B	0000400CH
VM-exit MSR-store count	000000111B	0000400EH
VM-exit MSR-load count	000001000B	00004010H
VM-entry controls	000001001B	00004012H
VM-entry MSR-load count	000001010B	00004014H
VM-entry interruption-information field	000001011B	00004016H

Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
VM-entry exception error code	000001100B	00004018H
VM-entry instruction length	000001101B	0000401AH
TPR threshold ¹	000001110B	0000401CH
Secondary processor-based VM-execution controls ²	000001111b	0000401EH
PLE_Gap ³	000010000b	00004020H
PLE_Window ³	000010001b	00004022H

NOTES:

1. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “PAUSE-loop exiting” VM-execution control.

B.3.2 32-Bit Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-9 enumerates the 32-bit read-only data fields.

Table B-9. Encodings for 32-Bit Read-Only Data Fields (0100_01xx_xxxx_xxx0B)

Field Name	Index	Encoding
VM-instruction error	000000000B	00004400H
Exit reason	000000001B	00004402H
VM-exit interruption information	000000010B	00004404H
VM-exit interruption error code	000000011B	00004406H
IDT-vectoring information field	000000100B	00004408H
IDT-vectoring error code	000000101B	0000440AH
VM-exit instruction length	000000110B	0000440CH
VM-exit instruction information	000000111B	0000440EH

B.3.3 32-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-10 enumerates the 32-bit guest-state fields.

Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest ES limit	000000000B	00004800H
Guest CS limit	000000001B	00004802H
Guest SS limit	000000010B	00004804H
Guest DS limit	000000011B	00004806H
Guest FS limit	000000100B	00004808H
Guest GS limit	000000101B	0000480AH
Guest LDTR limit	000000110B	0000480CH
Guest TR limit	000000111B	0000480EH

Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest GDTR limit	000001000B	00004810H
Guest IDTR limit	000001001B	00004812H
Guest ES access rights	000001010B	00004814H
Guest CS access rights	000001011B	00004816H
Guest SS access rights	000001100B	00004818H
Guest DS access rights	000001101B	0000481AH
Guest FS access rights	000001110B	0000481CH
Guest GS access rights	000001111B	0000481EH
Guest LDTR access rights	000010000B	00004820H
Guest TR access rights	000010001B	00004822H
Guest interruptibility state	000010010B	00004824H
Guest activity state	000010011B	00004826H
Guest SMBASE	000010100B	00004828H
Guest IA32_SYSENTER_CS	000010101B	0000482AH
VMX-preemption timer value ¹	000010111B	0000482EH

NOTES:

1. This field exists only on processors that support the 1-setting of the “activate VMX-preemption timer” VM-execution control.

The limit fields for GDTR and IDTR are defined to be 32 bits in width even though these fields are only 16-bits wide in the Intel 64 and IA-32 architectures. VM entry ensures that the high 16 bits of both these fields are cleared to 0.

B.3.4 32-Bit Host-State Field

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. There is only one such 32-bit field as given in Table B-11.

Table B-11. Encoding for 32-Bit Host-State Field (0100_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host IA32_SYSENTER_CS	000000000B	00004C00H

B.4 NATURAL-WIDTH FIELDS

A value of 3 in bits 14:13 of an encoding indicates a natural-width field. As noted in Section 24.11.2, each of these fields allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.4.1 Natural-Width Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-12 enumerates the natural-width control fields.

Table B-12. Encodings for Natural-Width Control Fields (0110_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
CRO guest/host mask	000000000B	00006000H

Table B-12. Encodings for Natural-Width Control Fields (0110_00xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
CR4 guest/host mask	000000001B	00006002H
CR0 read shadow	000000010B	00006004H
CR4 read shadow	000000011B	00006006H
CR3-target value 0	000000100B	00006008H
CR3-target value 1	000000101B	0000600AH
CR3-target value 2	000000110B	0000600CH
CR3-target value 3 ¹	000000111B	0000600EH

NOTES:

1. If a future implementation supports more than 4 CR3-target values, they will be encoded consecutively following the 4 encodings given here.

B.4.2 Natural-Width Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-13 enumerates the natural-width read-only data fields.

Table B-13. Encodings for Natural-Width Read-Only Data Fields (0110_01xx_xxxx_xxx0B)

Field Name	Index	Encoding
Exit qualification	000000000B	00006400H
I/O RCX	000000001B	00006402H
I/O RSI	000000010B	00006404H
I/O RDI	000000011B	00006406H
I/O RIP	000000100B	00006408H
Guest-linear address	000000101B	0000640AH

B.4.3 Natural-Width Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-14 enumerates the natural-width guest-state fields.

Table B-14. Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest CR0	000000000B	00006800H
Guest CR3	000000001B	00006802H
Guest CR4	000000010B	00006804H
Guest ES base	000000011B	00006806H
Guest CS base	000000100B	00006808H
Guest SS base	000000101B	0000680AH
Guest DS base	000000110B	0000680CH
Guest FS base	000000111B	0000680EH
Guest GS base	00001000B	00006810H

Table B-14. Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest LDTR base	000001001B	00006812H
Guest TR base	000001010B	00006814H
Guest GDTR base	000001011B	00006816H
Guest IDTR base	000001100B	00006818H
Guest DR7	000001101B	0000681AH
Guest RSP	000001110B	0000681CH
Guest RIP	000001111B	0000681EH
Guest RFLAGS	000010000B	00006820H
Guest pending debug exceptions	000010001B	00006822H
Guest IA32_SYSENTER_ESP	000010010B	00006824H
Guest IA32_SYSENTER_EIP	000010011B	00006826H
Guest IA32_S_CET ¹	000010100B	00006828H
Guest SSP ¹	000010101B	0000682AH
Guest IA32_INTERRUPT_SSP_TABLE_ADDR ¹	000010110B	0000682CH

NOTES:

1. This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.

The base-address fields for ES, CS, SS, and DS in the guest-state area are defined to be natural-width (with 64 bits on processors supporting Intel 64 architecture) even though these fields are only 32-bits wide in the Intel 64 architecture. VM entry ensures that the high 32 bits of these fields are cleared to 0.

B.4.4 Natural-Width Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-15 enumerates the natural-width host-state fields.

Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host CR0	000000000B	00006C00H
Host CR3	000000001B	00006C02H
Host CR4	000000010B	00006C04H
Host FS base	000000011B	00006C06H
Host GS base	000000100B	00006C08H
Host TR base	000000101B	00006C0AH
Host GDTR base	000000110B	00006C0CH
Host IDTR base	000000111B	00006C0EH
Host IA32_SYSENTER_ESP	000001000B	00006C10H
Host IA32_SYSENTER_EIP	000001001B	00006C12H
Host RSP	000001010B	00006C14H
Host RIP	000001011B	00006C16H
Host IA32_S_CET ¹	000001100B	00006C18H

Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Host SSP ¹	000001101B	00006C1AH
Host IA32_INTERRUPT_SSP_TABLE_ADDR ¹	000001110B	00006C1CH

NOTES:

1. This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.

36. 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 this chapter: Added Section 2.17.3 "MSRs Specific to 10th Generation Intel® Core™ Processors", and typo corrections/updates throughout the chapter.

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. The scope of an MSR defines the set of processors that access the same MSR with RDMSR and WRMSR. Thread-scope MSRs are unique to every logical processor. Core-scope MSRs are shared by the threads in the same core; similarly for module-scope, die-scope, and package-scope.

When a processor package contains a single die, die-scope and package-scope are synonymous. When a package contains multiple die, they are distinct.

NOTE

For information on hierarchical level types supported, refer to the CPUID Leaf 1FH definition for the actual level type numbers: "V2 Extended Topology Enumeration Leaf" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Also see Section 8.9.1, "Hierarchical Mapping of Shared Resources" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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_85H	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
06_7DH, 06_7EH	10th generation Intel® Core™ processors based on Ice Lake microarchitecture
06_A5H, 06_A6H	10th generation Intel® Core™ processors based on Comet Lake microarchitecture
06_66H	Intel® Core™ processors based on Cannon Lake microarchitecture
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture, 8th and 9th generation Intel® Core™ processors based on Coffee Lake microarchitecture, Intel® Xeon® E processors based on Coffee Lake microarchitecture
06_6AH, 06_6CH	Future Intel® Xeon® processors based on Ice Lake microarchitecture
06_55H	Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture, 2nd generation Intel® Xeon® Processor Scalable Family based on Cascade Lake product, and future Cooper Lake product
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

Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
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
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_86H	Intel® Atom™ processors based on Tremont Microarchitecture
06_7AH	Intel Atom processors based on Goldmont Plus Microarchitecture
06_5FH	Intel Atom processors based on Goldmont Microarchitecture (code name Denverton)
06_5CH	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

Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
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 **model-specific** 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 - 40000FFH 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 / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 2.23, “MSRs in Pentium Processors.”	Pentium Processor (05_01H)
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 2.23, “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 17.17, “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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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)	This register holds the APIC base address, permitting the relocation of the APIC memory map. See Section 10.4.4, "Local APIC Status and Location" and Section 10.4.5, "Relocating the Local APIC Registers".	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		
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.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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 a system executive that does not require SMX. BIOS must set this bit only when the CPUID function 1 returns the 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 field 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 re-configuration of SGX Launch Control via the 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	
		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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
48H	72	IA32_SPEC_CTRL	Speculation Control (R/W) The MSR bits are defined as logical processor scope. On some core implementations, the bits may impact sibling logical processors on the same core. This MSR has a value of 0 after reset and is unaffected by INIT# or SIPI#.	If any one of the enumeration conditions for defined bit field positions holds.
		0	Indirect Branch Restricted Speculation (IBRS). Restricts speculation of indirect branch.	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
		1	Single Thread Indirect Branch Predictors (STIBP). Prevents indirect branch predictions on all logical processors on the core from being controlled by any sibling logical processor in the same core.	If CPUID.(EAX=07H, ECX=0):EDX[27]=1
		2	Speculative Store Bypass Disable (SSBD) delays speculative execution of a load until the addresses for all older stores are known.	If CPUID.(EAX=07H, ECX=0):EDX[31]=1
		63:3	Reserved	
49H	73	IA32_PRED_CMD	Prediction Command (W0) Gives software a way to issue commands that affect the state of predictors.	If any one of the enumeration conditions for defined bit field positions holds.
		0	Indirect Branch Prediction Barrier (IBPB).	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
		63:1	Reserved	
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 9.1.1.6, "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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:32	It is recommended that this field be pre-loaded with zero prior to executing CPUID. If the field remains zero 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 && CPUID.(EAX=07H, ECX=0H):ECX[30]=1. Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1.
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.	
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.	
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
CFH	207	IA32_CORE_CAPABILITIES	IA32 Core Capabilities Register	If CPUID.(EAX=07H, ECX=0):EDX[30] = 1
		63:0	Reserved.	No architecturally defined bits.
E1H	225	IA32_UMWAIT_CONTROL	UMWAIT Control (R/W)	
		0	CO.2 is not allowed by the OS. Value of "1" means all CO.2 requests revert to CO.1.	
		1	Reserved	
		31:2	Determines the maximum time in TSC-quanta that the processor can reside in either CO.1 or CO.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.	
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	CO_MCNT: CO TSC Frequency Clock Count Increments at fixed interval (relative to TSC freq.) when the logical processor is in CO. 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	CO_ACNT: CO Actual Frequency Clock Count Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in CO. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) See 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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		8	Fixed range MTRRs are supported when set.	
		9	Reserved	
		10	wC Supported when set.	
		11	SMRR Supported when set.	
		12	PRMRR supported when set.	
		63:13	Reserved	
10AH	266	IA32_ARCH_CAPABILITIES	Enumeration of Architectural Features (RO)	If CPUID.(EAX=07H, ECX=0):EDX[29]=1
		0	RDCL_NO: The processor is not susceptible to Rogue Data Cache Load (RDCL).	
		1	IBRS_ALL: The processor supports enhanced IBRS.	
		2	RSBA: The processor supports RSB Alternate. Alternative branch predictors may be used by RET instructions when the RSB is empty. SW using retpoline may be affected by this behavior.	
		3	SKIP_L1DFL_VMENTRY: A value of 1 indicates the hypervisor need not flush the L1D on VM entry.	
		4	SSB_NO: Processor is not susceptible to Speculative Store Bypass.	
		5	MDS_NO: Processor is not susceptible to Microarchitectural Data Sampling (MDS).	
		6	IF_PSCCHANGE_MC_NO: The processor is not susceptible to a machine check error due to modifying the size of a code page without TLB invalidation.	
		7	TSX_CTRL: If 1, indicates presence of IA32_TSX_CTRL MSR.	
		8	TAA_NO: If 1, processor is not affected by TAA.	
		63:9	Reserved	
10BH	267	IA32_FLUSH_CMD	Flush Command (WO) Gives software a way to invalidate structures with finer granularity than other architectural methods.	If any one of the enumeration conditions for defined bit field positions holds.
		0	L1D_FLUSH: Writeback and invalidate the L1 data cache.	If CPUID.(EAX=07H, ECX=0):EDX[28]=1
		63:1	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
122H	290	IA32_TSX_CTRL	IA32_TSX_CTRL	Thread scope. Not architecturally serializing. Available when CPUID.ARCH_CAP(EAX=7H, ECX = 0);EDX[29] = 1 and IA32_ARCH_CAPABILITIES. bit 7 = 1.
		0	RTM_DISABLE: When set to 1, XBEGIN will always abort with EAX code 0.	
		1	TSX_CPUID_CLEAR: When set to 1, CPUID.07H.EBX.RTM [bit 11] and CPUID.07H.EBX.HLE [bit 4] report 0. When set to 0 and the SKU supports TSX, these bits will return 1.	
		63:2	Reserved	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector.	
		31:16	Not used.	Can be read and written.
		63:32	Not used.	Writes ignored; reads return zero.
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
		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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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 supports 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	MCIIP. 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_PERFEVTSELO (PERFEVTSELO)	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.	
		18	Edge: Enables edge detection if set.	
		19	PC: Enables pin control.	
		20	INT: Enables interrupt on counter overflow.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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	Current Performance Status (RO) See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Current performance State Value.	
		63:16	Reserved	
199H	409	IA32_PERF_CTL	Performance Control MSR (R/W) Software makes a request for a new Performance state (P-State) by writing this MSR. See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	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 14.8.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	Extended On-Demand Clock Modulation Duty Cycle.	If CPUID.06H:EAX[5] = 1
		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 14.8.2, "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 14.8.2, "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/WCO)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WCO)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WCO)	If CPUID.01H:ECX[8] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #2 log (R/WCO)	If CPUID.01H:ECX[8] = 1
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WCO)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WCO)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WCO)	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.	OF_OH
		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.	OF_OH
		6:4	Reserved	
		7	Performance Monitoring Available (R) 1 = Performance monitoring enabled. 0 = Performance monitoring disabled.	OF_OH
		10:8	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / 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 / 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 / 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 14.9, "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/WCO)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status Log (R/WCO)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 Log (R/WCO)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 Log (R/WCO)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation Log (R/WCO)	
		15:12	Reserved	
		22:16	Pkg Digital Readout (RO)	
		25:23	Reserved	
26	Hardware Feedback Interface Structure Change Status	If CPUID.06H:EAX.[19] = 1		
63:27	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 14.9, "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			

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	
		25	Hardware Feedback Interrupt Enable	If CPUID.06H:EAX.[19] = 1
		63:26	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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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.	
		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 IA32_MTRRCAP[7:0] > 0
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If IA32_MTRRCAP[7:0] > 0
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If IA32_MTRRCAP[7:0] > 1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If IA32_MTRRCAP[7:0] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If IA32_MTRRCAP[7:0] > 2
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If IA32_MTRRCAP[7:0] > 2
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If IA32_MTRRCAP[7:0] > 3
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If IA32_MTRRCAP[7:0] > 3
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If IA32_MTRRCAP[7:0] > 4
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If IA32_MTRRCAP[7:0] > 4
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If IA32_MTRRCAP[7:0] > 5
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If IA32_MTRRCAP[7:0] > 5
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If IA32_MTRRCAP[7:0] > 6
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If IA32_MTRRCAP[7:0] > 6
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If IA32_MTRRCAP[7:0] > 7
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If IA32_MTRRCAP[7:0] > 7
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
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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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_MCO_CTL2	MSR to enable/disable CMCI capability for bank 0. (R/W) See Section 15.3.2.5, "IA32_MCi_CTL2 MSRs".	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_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2
283H	643	IA32_MC3_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
287H	647	IA32_MC7_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
28CH	652	IA32_MC12_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17
292H	658	IA32_MC18_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
296H	662	IA32_MC22_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
29BH	667	IA32_MC27_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) Same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) Same fields as IA32_MCO_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	
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved	
309H	777	IA32_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	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core.	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref.	If CPUID.0AH: EDX[4:0] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
345H	837	IA32_PERF_CAPABILITIES	Read Only MSR that enumerates the existence of performance monitoring features. (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.	
		14	Reserved	
		15	1: Performance metrics available.	
		16	1: PEBS output will be written into the Intel PT trace stream.	If CPUID.0x7.0.EBX[25]=1
	63:17	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 1 to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1 to count while CPL > 0.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / 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
		47:35	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		48	Ovf_PERF_METRICS: If this bit is set, it indicates that PERF_METRIC counter has overflowed and a PMI is triggered; however, an overflow of fixed counter 3 should normally happen first. If this bit is clear no overflow occurred.	
		54:49	Reserved	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer that was completely filled.	If (CPUID.(EAX=07H, ECX=0);EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved	
		58	LBR_Frz. LBRs are frozen due to: <ul style="list-style-type: none"> ▪ 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: <ul style="list-style-type: none"> ▪ 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 have 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 the 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_CTRO	If CPUID.0AH: EDX[4:0] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2
		47:35	Reserved	
		48	EN_PERF_METRICS: If this bit is set and fixed counter 3 is effectively enabled, built-in performance metrics are enabled.	
		63:49	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
		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.	O6_2EH
		62	Set 1 to Clear OvfBuf bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set 1 to 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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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
		47:35	Reserved	
		48	RESET_OVF_PERF_METRICS: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
		54:49	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
		63	Set 1 to 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
		47:35	Reserved	
		48	SET_OVF_PERF_METRICS: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
		54:49	Reserved	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.0AH: EAX[7:0] > 3
		57:56	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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 that core perfmon interface is in use. (RO)	If CPUID.0AH: EAX[7:0] > 3
		0	IA32_PERFEVTSELO 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+1	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)	
		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_MCO_CTL	MCO_CTL	If IA32_MCG_CAP.CNT > 0
401H	1025	IA32_MCO_STATUS	MCO_STATUS	If IA32_MCG_CAP.CNT > 0
402H	1026	IA32_MCO_ADDR ¹	MCO_ADDR	If IA32_MCG_CAP.CNT > 0
403H	1027	IA32_MCO_MISC	MCO_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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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	1112	IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H	1113	IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH	1114	IA32_MC22_ADDR ¹	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH	1115	IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH	1116	IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH	1117	IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH	1118	IA32_MC23_ADDR ¹	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH	1119	IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H	1120	IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H	1121	IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H	1122	IA32_MC24_ADDR ¹	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H	1123	IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H	1124	IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H	1125	IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H	1126	IA32_MC25_ADDR ¹	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H	1127	IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H	1128	IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H	1129	IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26
46AH	1130	IA32_MC26_ADDR ¹	MC26_ADDR	If IA32_MCG_CAP.CNT >26
46BH	1131	IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT >26
46CH	1132	IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT >27
46DH	1133	IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT >27
46EH	1134	IA32_MC27_ADDR ¹	MC27_ADDR	If IA32_MCG_CAP.CNT >27
46FH	1135	IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT >27
470H	1136	IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT >28
471H	1137	IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT >28
472H	1138	IA32_MC28_ADDR ¹	MC28_ADDR	If IA32_MCG_CAP.CNT >28
473H	1139	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_CTL	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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
482H	1154	IA32_VMX_PROCBASED_CTL5	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_CTL5	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_CTL5	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
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_CTL52	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_CTL5[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_CTL5[63] && (IA32_VMX_PROCBASED_CTL52[33] IA32_VMX_PROCBASED_CTL52[37]))
48DH	1165	IA32_VMX_TRUE_PINBASED_CTL5	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])

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL5	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_CTL5	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_CTL5	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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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	Allows software to signal some MCEs to only a single logical processor in the system. (R/W) See Section 15.3.1.4, "IA32_MCG_EXT_CTL MSR".	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	
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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		1	CYCEn	If (CPUID.(EAX=07H, ECX=0);EBX[1] = 1)
		2	OS	
		3	User	
		4	PwrEvtEn	If (CPUID.(EAX=07H, ECX=1);EBX[5] = 1)
		5	FUPonPTW	If (CPUID.(EAX=07H, ECX=1);EBX[4] = 1)
		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	PTWEn	If (CPUID.(EAX=07H, ECX=1);EBX[4] = 1)
		13	BranchEn	
		17:14	MTCFreq	If (CPUID.(EAX=07H, ECX=0);EBX[3] = 1)
		18	Reserved, must be zero.	
		22:19	CYCThresh	If (CPUID.(EAX=07H, ECX=0);EBX[1] = 1)
		23	Reserved, must be zero.	
		27:24	PSBFreq	If (CPUID.(EAX=07H, ECX=0);EBX[1] = 1)
		31:28	Reserved, must be zero.	
		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)
		55:48	Reserved, must be zero.	
		56	InjectPsbPmiOnEnable	If (CPUID.(EAX=07H, ECX=1);EBX[6] = 1)
		63:57	Reserved, must be zero.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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	
		6	PendPSB	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
		7	PendToPAPMI	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
		31:8	Reserved, must be zero.	
		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	
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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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 18.6.3.4, "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.	
6A0H	1696	IA32_U_CET	Configure User Mode CET (R/W)	Bits 1:0 are defined if CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1. Bits 5:2 and bits 63:10 are defined if CPUID.(EAX=07H, ECX=0H):EDX.CET_IBT[20] = 1.
		0	SH_STK_EN: When set to 1, enable shadow stacks at CPL3.	
		1	WR_SHSTK_EN: When set to 1, enables the WRSSD/WRSSQ instructions.	
		2	ENDBR_EN: When set to 1, enables indirect branch tracking.	
		3	LEG_IW_EN: Enable legacy compatibility treatment for indirect branch tracking.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		4	NO_TRACK_EN: When set to 1, enables use of no-track prefix for indirect branch tracking.	
		5	SUPPRESS_DIS: When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.	
		9:6	Reserved; must be zero.	
		10	SUPPRESS: When set to 1, indirect branch tracking is suppressed. This bit can be written to 1 only if TRACKER is written as IDLE.	
		11	TRACKER: Value of the indirect branch tracking state machine. Values: IDLE (0), WAIT_FOR_ENDBRANCH(1).	
		63:12	EB_LEG_BITMAP_BASE: Linear address bits 63:12 of a legacy code page bitmap used for legacy compatibility when indirect branch tracking is enabled. If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	
6A2H	1698	IA32_S_CET	Configure Supervisor Mode CET (R/W)	See IA32_U_CET (6A0H) for reference; similar format.
6A4H	1700	IA32_PLO_SSP	Linear address to be loaded into SSP on transition to privilege level 0. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
6A5H	1701	IA32_PL1_SSP	Linear address to be loaded into SSP on transition to privilege level 1. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A6H	1702	IA32_PL2_SSP	Linear address to be loaded into SSP on transition to privilege level 2. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A7H	1703	IA32_PL3_SSP	Linear address to be loaded into SSP on transition to privilege level 3. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. The linear address written must be aligned to 8 bytes and bits 2:0 must be 0 (hardware requires bits 1:0 to be 0).	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6A8H	1704	IA32_INTERRUPT_SSP_TABLE_ADDR	Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W) This MSR is not present on processors that do not support Intel 64 architecture. This field cannot represent a non-canonical address.	If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
6E0H	1760	IA32_TSC_DEADLINE	TSC Target of Local APIC's TSC Deadline Mode (R/W)	If CPUID.01H:ECX.[24] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
6E1H	1761	IA32_PKRS	Specifies the PK permissions associated with each protection domain for supervisor pages (R/W)	If CPUID.(EAX=07H, ECX=0H):ECX.PKS [31] = 1
		31:0	For domain i (i between 0 and 15), bits 2i and 2i+1 contain the AD and WD permissions, respectively.	
		63:32	Reserved.	
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
		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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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
		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	
775H	1909	IA32_PECI_HWP_REQUEST_INFO	Package	Enable Peci HWP Override Values (RO)
		7:0		Minimum Performance (MINIMUM_PERFORMANCE): Used by OS to read the latest value of Peci minimum performance input. Default value is 0.
		15:8		Maximum Performance (MAXIMUM_PERFORMANCE): Used by OS to read the latest value of Peci maximum performance input. Default value is 0.
		23:16		Reserved.
		31:24		Energy Performance Preference (ENERGY_PERFORMANCE_PREFERENCE): Used by OS to read the latest value of Peci Energy Performance Preference input. Default value is 0.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		59:32		Reserved.
		60		EPP PECC Override (EPP_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Energy Performance Preference input. If set to '1', PECC is overriding the Energy Performance Preference input. If clear (0), OS has control over Energy Performance Preference input. Default value is 0.
		61		Reserved.
		62		Max PECC Override (MAX_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Maximum Performance input. If set to '1', PECC is overriding the Maximum Performance input. If clear (0), OS has control over Maximum Performance input. Default value is 0.
		63		Min PECC Override (MIN_PECC_OVERRIDE): Indicates whether PECC is currently overriding the Minimum Performance input. If set to '1', PECC is overriding the Minimum Performance input. If clear (0), OS has control over Minimum Performance input. Default value is 0.
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/WCO) See Section 14.4.5, "HWP Feedback".	If CPUID.06H:EAX.[7] = 1
		1	Reserved	
		2	Excursion_To_Minimum (R/WCO) See Section 14.4.5, "HWP Feedback".	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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)
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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. Attempts to write to reserved bits result in a #GP(0).	
C82H	3202	IA32_L2_QOS_CFG	L2 QOS Configuration (R/W)	If (CPUID.(EAX=10H, ECX=2):ECX.[2] = 1)
		0	Enable (R/W) Set 1 to enable L2 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode.	
		63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).	
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.	
		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 an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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	3216 - 3471	Reserved MSR Address Space for CAT Mask Registers	See Section 17.19.4.1, "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	3344 - 3407	Reserved MSR Address Space for L2 CAT Mask Registers	See Section 17.19.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	
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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		1	BNDPRESERVE: Preserve the bounds registers for near branch instructions in the absence of the BND prefix.	
		11:2	Reserved, must be zero.	
		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)	
		10:9	Reserved.	
		11	CET_U State (R/W)	
		12	CET_S State (R/W)	
		13	HDC State (R/W)	
		63:14	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/O)	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
17D0H	6096	IA32_HW_FEEDBACK_PTR	Hardware Feedback Interface Pointer	If CPUID.06H:EAX.[19] = 1
		0	Valid (R/W) When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.	
		11:1	Reserved	
		(MAXPHYADDR-1):12	ADDR (R/W) Physical address of the page frame of the first page of the hardware feedback interface structure.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:MAXPHYADDR	Reserved	
17D1H	6097	IA32_HW_FEEDBACK_CONFIG	Hardware Feedback Interface Configuration	If CPUID.06H:EAX.[19] = 1
		0	Enable (R/W) When set to 1, enables the hardware feedback interface.	
		63:1	Reserved	
4000_0000H - 4000_00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSRs 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) Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.	If CPUID.80000001:EDX.[29] = 1
C000_0083H		IA32_CSTAR	IA-32e Mode System Call Target Address (R/W) Not used, as the SYSCALL instruction is not recognized in compatibility mode.	If CPUID.80000001:EDX.[29] = 1
C000_0084H		IA32_FMASK	System Call Flag Mask (R/W)	If CPUID.80000001:EDX.[29] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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 or CPUID.(EAX=7,ECX=0):ECX[bit 22] = 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_MCI_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 / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Unique	See Section 2.23, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination.” and Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "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
		1		Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement 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 processors implement R/W.
		4		Address Parity Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		5		Reserved
		6		Reserved
		7		BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled Note: Not all processors implement R/W.
		8		Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled
9		Execute BIST (R/O) 1 = Enabled; 0 = Disabled		

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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)
3AH	58	MSR_FEATURE_CONTROL	Unique	Control Features in Intel 64 Processor (R/W) See Table 2-2.
		3	Unique	SMRR Enable (R/WL) When this bit is set and the lock bit is set, this 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: <ul style="list-style-type: none"> ▪ 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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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
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		<ul style="list-style-type: none"> ▪ 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)

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	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 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		<ul style="list-style-type: none"> ▪ 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) <p>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.</p>
		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)
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	Global Machine Check Status

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	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_PERFEVTSELO	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	Current performance status. See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".
		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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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	Unique	Thermal Monitor 2 Control
		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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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.
				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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		20	Shared	Enhanced Intel SpeedStep Technology Select Lock (R/WO) When set, this bit causes the following bits to become read-only: <ul style="list-style-type: none"> 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
		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 cleared (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 the power-on default value is 1, IDA is available in the processor. If the power-on default value is 0, IDA is not available.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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/W) 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/W) 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	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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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.
30AH	778	IA32_FIXED_CTR1	Unique	Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Unique	Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.
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.
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	MSR_PERF_GLOBAL_CTRL	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, “Global Counter Control Facilities.”
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Unique	See Section 18.6.2.2, “Global Counter Control Facilities.”
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, “Processor Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MCO_CTL	Unique	See Section 15.3.2.1, “IA32_MCI_CTL MSRs.”
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, “IA32_MCI_STATUS MSRs.”
402H	1026	IA32_MCO_ADDR	Unique	See Section 15.3.2.3, “IA32_MCI_ADDR MSRs.” The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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 ADDR_V 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 ADDR_V 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 ADDR_V 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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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 ADDR_V 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	Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISC_V flag in the IA32_MCi_STATUS register is set.
414H	1044	IA32_MC5_CTL	Unique	Machine Check Error Reporting Register: Controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
415H	1045	IA32_MC5_STATUS	Unique	Machine Check Error Reporting Register: Contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	IA32_MC5_ADDR	Unique	Machine Check Error Reporting Register: Contains the address of the code or data memory location that produced the machine-check error if the ADDR_V flag in the IA32_MCi_STATUS register is set.
417H	1047	IA32_MC5_MISC	Unique	Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISC_V flag in the IA32_MCi_STATUS register is set.
419H	1045	IA32_MC6_STATUS	Unique	Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 23.
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_CTL	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_CTL	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_CTL	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
484H	1156	IA32_VMX_ENTRY_CTL5	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_CRO_FIXED0	Unique	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
487H	1159	IA32_VMX_CRO_FIXED1	Unique	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
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_CTL52	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 18.6.3.4, "Debug Store (DS) Mechanism."
107C H	67532	MSR_EMON_L3_CTR_CTL0	Unique	GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107C H	67533	MSR_EMON_L3_CTR_CTL1	Unique	GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107C H	67534	MSR_EMON_L3_CTR_CTL2	Unique	GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
107CF H	67535	MSR_EMON_L3_CTR_CTL3	Unique	GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H	67536	MSR_EMON_L3_CTR_CTL4	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H	67537	MSR_EMON_L3_CTR_CTL5	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H	67538	MSR_EMON_L3_CTR_CTL6	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H	67539	MSR_EMON_L3_CTR_CTL7	Unique	FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D8 H	67544	MSR_EMON_L3_GL_CTL	Unique	L3/FSB Common Control Register (R/W) Applies 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.
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 / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Shared	See Section 2.23, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination.” and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, “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
		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

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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.
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: <ul style="list-style-type: none"> ▪ 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.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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.
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		<ul style="list-style-type: none"> ▪ 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

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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	Control Register 3 Used to configure the L2 Cache.
		0		L2 Hardware Enabled (RO) 1 = Indicates the L2 is hardware-enabled. 0 = Indicates 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	Global Machine Check Status
		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.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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_PERFEVTSELO	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	Performance Status
		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	Thermal Monitor 2 Control
		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.
		63:17		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.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		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.
				When this bit is cleared (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

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
		20	Shared	Enhanced Intel SpeedStep Technology Select Lock (R/W) When set, this bit causes the following bits to become read-only: <ul style="list-style-type: none"> ▪ 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.
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.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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.
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 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0 (R/W)
400H	1024	IA32_MCO_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
401H	1025	IA32_MCO_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRS." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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 ADDR_V 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 ADDR_V 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."
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 ADDR_V 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_CTL	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
482H	1154	IA32_VMX_PROCBASED_CTL5	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_CTL5	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_CTL5	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_CRO_FIXED0	Unique	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
487H	1159	IA32_VMX_CRO_FIXED1	Unique	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
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_CTL52	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 18.6.3.4, "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.

Table 2-4. MSRs in 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address		Register Name / Bit Fields	Shared/ Unique	Bit Description
Hex	Dec			
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 / Bit Fields	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.
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

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.17, "Time-Stamp Counter," and 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
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
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 .

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 sets 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 instructions can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved
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	Global Machine Check Status

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	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.
19BH	411	IA32_THERM_INTERRUPT	Core	Thermal Interrupt Control (R/W) See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
19CH	412	IA32_THERM_STATUS	Core	Thermal Monitor Status (R/W) See Table 2-2.
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The default thermal throttling or PROCHOT# activation temperature in degrees 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/W) 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/W) 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

Register Address		Register Name / Bit Fields	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 18.6.2.2, "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

Register Address		Register Name / Bit Fields	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_MCO_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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 ADDR_V 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 ADDR_V 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 ADDR_V 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."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V 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_CTLX	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_CTLX	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_CTLX	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_CTLX	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."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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."
48BH	1163	IA32_VMX_PROCBASED_CTL2	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_CTL2	Core	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL2	Core	Capability Reporting Register of Primary Processor-based VM-Execution Flex Controls (R/O) See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTL2	Core	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTL2	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 18.6.3.4, "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.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom® Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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 / Bit Fields	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)

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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: <ul style="list-style-type: none"> ▪ 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.
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.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information: Contains power management and other model specific features enumeration. See http://biosbits.org .

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.
		63:16		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: C0 (no package C-state 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, locks bits 15:0 of this register until next reset.
		63:16		Reserved
11EH	281	MSR_BBL_CR_CTL3	Module	Control Register 3 Used to configure the L2 Cache.
		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.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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		

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		38	Module	<p>Turbo Mode Disable (R/W)</p> <p>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 cleared (CPUID.06H: EAX[1]=0).</p> <p>When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled.</p> <p>Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</p>
		63:39		Reserved
1C8H	456	MSR_LBR_SELECT	Core	Last Branch Record Filtering Select Register (R/W) See Section 17.9.2, "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	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 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "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.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	Module C6 Residency Counter (R/O) 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 / Bit Fields	Scope	Bit Description
Hex	Dec			
CDH	205	MSR_FSB_FREQ	Module	Scaleable Bus Speed(R0) This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture.
		2:0		<ul style="list-style-type: none"> ▪ 100B: 080.0 MHz ▪ 000B: 083.3 MHz ▪ 001B: 100.0 MHz ▪ 010B: 133.3 MHz ▪ 011B: 116.7 MHz
		63:3		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "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

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 / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 14.10.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1 (R/W) See Section 14.10.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1 (R/W) See Section 14.10.3, "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 14.10.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/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		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-9. Specific MSRs Supported by Intel Atom® Processor E3000 Series (Contd.)with CPUID Signature 06_37H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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		Register Name / Bit Fields	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
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.
		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 14.10.1, "RAPL Interfaces."

Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series (Contd.)with CPUID Signature

Register Address		Register Name / Bit Fields	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) See Section 14.10.3, "Package RAPL Domain."
66EH	1646	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameter (R/O)
		14:0		Thermal Spec Power (R/O) The unsigned integer value is the equivalent of the 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

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.

Table 2-11. MSRs in Intel Atom® Processors Based on the Airmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		3:0		<ul style="list-style-type: none"> ▪ 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, locks 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.)

Register Address		Register Name / Bit Fields	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_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W)
		14:0		PPO Power Limit #1 (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains" and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1 (R/W) See Section 14.10.4, "PPO/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 PPO_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 INTEL ATOM® PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Intel Atom processors based on the Goldmont microarchitecture support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID signature with DisplayFamily_DisplayModel of 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 Intel Atom® Processors Based on the Goldmont Microarchitecture

Register Address		Register Name / Bit Fields	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 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)
		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	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		30	Package	Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, 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, locks bits 15:0 of this register until next reset.
		63:16		Reserved
17DH	381	MSR_SMM_MCA_CAP	Core	Enhanced SMM Capabilities (SMM-R0) Reports SMM capability enhancement. Accessible only while in SMM.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
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.		

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		37:35		Reserved
		38	Package	<p>Turbo Mode Disable (R/W)</p> <p>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).</p> <p>When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled.</p> <p>Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</p>
		63:39		Reserved
1A4H	420	MSR_MISC_FEATURE_CONTROL		Miscellaneous Feature Control (R/W)
		0	Core	<p>L2 Hardware Prefetcher Disable (R/W)</p> <p>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</p>
		1		Reserved
		2	Core	<p>DCU Hardware Prefetcher Disable (R/W)</p> <p>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</p>
		63:3		Reserved
1AAH	426	MSR_MISC_PWR_MGMT	Package	<p>Miscellaneous Power Management Control</p> <p>Various model specific features enumeration. See http://biosbits.org.</p>
		0		<p>EIST Hardware Coordination Disable (R/W)</p> <p>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.</p>
		21:1		Reserved
		22		<p>Thermal Interrupt Coordination Enable (R/W)</p> <p>If set, then thermal interrupt on one core is routed to all cores.</p>
		63:23		Reserved

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 the number of active cores is less than or equal to the Group 0 threshold.
		15:8	Package	Maximum Ratio Limit for Active Cores in Group 1 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 1 threshold, and greater than the Group 0 threshold.
		23:16	Package	Maximum Ratio Limit for Active Cores in Group 2 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 2 threshold, and greater than the Group 1 threshold.
		31:24	Package	Maximum Ratio Limit for Active Cores in Group 3 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 3 threshold, and greater than the Group 2 threshold.
		39:32	Package	Maximum Ratio Limit for Active Cores in Group 4 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 4 threshold, and greater than the Group 3 threshold.
		47:40	Package	Maximum Ratio Limit for Active Cores in Group 5 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 5 threshold, and greater than the Group 4 threshold.
		55:48	Package	Maximum Ratio Limit for Active Cores in Group 6 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 6 threshold, and greater than the Group 5 threshold.
		63:56	Package	Maximum Ratio Limit for Active Cores in Group 7 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 7 threshold, and greater than the Group 6 threshold.
1AEH	430	MSR_TURBO_GROUP_CORECNT	Package	Group Size of Active Cores for Turbo Mode Operation (RW) Writes of 0 threshold is ignored.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0	Package	Group 0 Core Count Threshold Maximum number of active cores to operate under the Group 0 Max Turbo Ratio limit.
		15:8	Package	Group 1 Core Count Threshold Maximum number of active cores to operate under the Group 1 Max Turbo Ratio limit. Must be greater than the Group 0 Core Count.
		23:16	Package	Group 2 Core Count Threshold Maximum number of active cores to operate under the Group 2 Max Turbo Ratio limit. Must be greater than the Group 1 Core Count.
		31:24	Package	Group 3 Core Count Threshold Maximum number of active cores to operate under the Group 3 Max Turbo Ratio limit. Must be greater than the Group 2 Core Count.
		39:32	Package	Group 4 Core Count Threshold Maximum number of active cores to operate under the Group 4 Max Turbo Ratio limit. Must be greater than the Group 3 Core Count.
		47:40	Package	Group 5 Core Count Threshold Maximum number of active cores to operate under the Group 5 Max Turbo Ratio limit. Must be greater than the Group 4 Core Count.
		55:48	Package	Group 6 Core Count Threshold Maximum number of active cores to operate under the Group 6 Max Turbo Ratio limit. Must be greater than the Group 5 Core Count.
		63:56	Package	Group 7 Core Count Threshold Maximum number of active cores to operate under the Group 7 Max Turbo Ratio limit. Must be greater than the 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 17.9.2, "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

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 (W) 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 (W) 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."
		0		Ovf_PMC0

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	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	Core	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		
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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
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 ADDR_V 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-RW0) 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 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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 OFFFH. 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 41.1.1.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1		Reserved
		23:16		SGX_SVN_SINIT See Section 41.1.1.3, "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, must be zero.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		BranchEn
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDRO_CFG
		39:36		ADDR1_CFG
		63:40		Reserved, must be zero.
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, must be zero.
		48:32		PacketByteCnt
		63:49		Reserved, must be zero.
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_ADDRO_A	Core	Region 0 Start Address (R/W)
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDRO_B	Core	Region 0 End Address (R/W)
		63:0		See Table 2-2.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 14.10.1, "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
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 2-20 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 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
60BH	1547	MSR_PKG_C2_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 a transition to a 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 2-20 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-state management.
		63:16		Reserved
60CH	1548	MSR_PKG_C2_IRTL2	Package	Package C7 Interrupt Response Limit 2 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a 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 2-20 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-state 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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W)
		14:0		Thermal Spec Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		15		Reserved
		30:16		Minimum Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		31		Reserved
		46:32		Maximum Power (R/W) See Section 14.10.3, "Package RAPL Domain."
		47		Reserved
		54:48		Maximum Time Window (R/W) Specified by $2^Y * (1.0 + Z/4.0) * \text{Time_Unit}$, 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 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
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.
		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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W)
		7:0		MAX_NON_TURBO_RATIO (R/W/L) System BIOS can program this field.
		30:8		Reserved
		31		TURBO_ACTIVATION_RATIO_Lock (R/W/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 Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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: <ul style="list-style-type: none"> ▪ 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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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 19 From 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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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 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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
6DOH	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 19 To 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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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)
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)
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)

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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)
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.

Table 2-12. MSRs in Intel Atom® Processors Based on the Goldmont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 INTEL ATOM® PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Intel Atom processors based on the Goldmont Plus microarchitecture support MSRs listed in Table 2-6, Table 2-12 and Table 2-13. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_7AH; see Table 2-1. For an MSR listed in Table 2-13 that also appears in the model-specific tables of prior generations, Table 2-13 supersedes prior generation tables.

In the Goldmont Plus 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 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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)
		17		SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Valid if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		SGX global functions enable (R/WL)
		63:19		Reserved
8CH	140	IA32_SGXLEPUBKEYHASH0	Core	See Table 2-2.

Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
8DH	141	IA32_SGXLEPUBKEYHASH1	Core	See Table 2-2.
8EH	142	IA32_SGXLEPUBKEYHASH2	Core	See Table 2-2.
8FH	143	IA32_SGXLEPUBKEYHASH3	Core	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	(R/W) See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0.
		1		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC1.
		2		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC2.
		3		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC3.
		31:4		Reserved
		32		Enable PEBS trigger and recording for IA32_FIXED_CTR0.
		33		Enable PEBS trigger and recording for IA32_FIXED_CTR1.
		34		Enable PEBS trigger and recording for IA32_FIXED_CTR2.
		63:35		Reserved
570H	1392	IA32_RTIT_CTL	Core	Trace Control Register (R/W)
		0		TraceEn
		1		CYCEn
		2		OS
		3		User
		4		PwrEvtEn
		5		FUPonPTW
		6		FabricEn
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
		11		DisRETC
		12		PTWEn
13		BranchEn		

Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDRO_CFG
		39:36		ADDR1_CFG
		63:40		Reserved, must be zero.
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Core	Last Branch Record 0 From IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The From_IP part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> Last Branch Record Stack TOS at 1C9H. Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."
681H - 69FH	1665 - 1695	MSR_LASTBRANCH_i_FROM_IP	Core	Last Branch Record <i>i</i> From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP; <i>i</i> = 1-31.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Core	Last Branch Record 0 To IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The To_IP part of the stack contains pointers to the Destination instruction. See also: <ul style="list-style-type: none"> Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."
6C1H - 6DFH	1729 - 1759	MSR_LASTBRANCH_i_TO_IP	Core	Last Branch Record <i>i</i> To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP; <i>i</i> = 1-31.
DC0H	3520	MSR_LASTBRANCH_INFO_0	Core	Last Branch Record 0 Additional Information (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. This part of the stack contains flag and elapsed cycle information. See also: <ul style="list-style-type: none"> Last Branch Record Stack TOS at 1C9H. Section 17.9.1, "LBR Stack."
DC1H	3521	MSR_LASTBRANCH_INFO_1	Core	Last Branch Record 1 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC2H	3522	MSR_LASTBRANCH_INFO_2	Core	Last Branch Record 2 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC3H	3523	MSR_LASTBRANCH_INFO_3	Core	Last Branch Record 3 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DC4H	3524	MSR_LASTBRANCH_INFO_4	Core	Last Branch Record 4 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC5H	3525	MSR_LASTBRANCH_INFO_5	Core	Last Branch Record 5 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC6H	3526	MSR_LASTBRANCH_INFO_6	Core	Last Branch Record 6 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC7H	3527	MSR_LASTBRANCH_INFO_7	Core	Last Branch Record 7 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC8H	3528	MSR_LASTBRANCH_INFO_8	Core	Last Branch Record 8 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DC9H	3529	MSR_LASTBRANCH_INFO_9	Core	Last Branch Record 9 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCAH	3530	MSR_LASTBRANCH_INFO_10	Core	Last Branch Record 10 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCBH	3531	MSR_LASTBRANCH_INFO_11	Core	Last Branch Record 11 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCCH	3532	MSR_LASTBRANCH_INFO_12	Core	Last Branch Record 12 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCDH	3533	MSR_LASTBRANCH_INFO_13	Core	Last Branch Record 13 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCEH	3534	MSR_LASTBRANCH_INFO_14	Core	Last Branch Record 14 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DCFH	3535	MSR_LASTBRANCH_INFO_15	Core	Last Branch Record 15 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD0H	3536	MSR_LASTBRANCH_INFO_16	Core	Last Branch Record 16 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD1H	3537	MSR_LASTBRANCH_INFO_17	Core	Last Branch Record 17 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD2H	3538	MSR_LASTBRANCH_INFO_18	Core	Last Branch Record 18 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD3H	3539	MSR_LASTBRANCH_INFO_19	Core	Last Branch Record 19 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD4H	3520	MSR_LASTBRANCH_INFO_20	Core	Last Branch Record 20 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD5H	3521	MSR_LASTBRANCH_INFO_21	Core	Last Branch Record 21 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD6H	3522	MSR_LASTBRANCH_INFO_22	Core	Last Branch Record 22 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

Table 2-13. MSRs in Intel Atom® Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DD7H	3523	MSR_LASTBRANCH_INFO_23	Core	Last Branch Record 23 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD8H	3524	MSR_LASTBRANCH_INFO_24	Core	Last Branch Record 24 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DD9H	3525	MSR_LASTBRANCH_INFO_25	Core	Last Branch Record 25 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDAH	3526	MSR_LASTBRANCH_INFO_26	Core	Last Branch Record 26 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDBH	3527	MSR_LASTBRANCH_INFO_27	Core	Last Branch Record 27 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDCH	3528	MSR_LASTBRANCH_INFO_28	Core	Last Branch Record 28 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDDH	3529	MSR_LASTBRANCH_INFO_29	Core	Last Branch Record 29 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDEH	3530	MSR_LASTBRANCH_INFO_30	Core	Last Branch Record 30 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.
DDFH	3531	MSR_LASTBRANCH_INFO_31	Core	Last Branch Record 31 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.

See Table 2-6, Table 2-12 and Table 2-13 for MSR definitions applicable to processors with CPUID signature 06_7AH.

2.7 MSRS IN INTEL ATOM® PROCESSORS BASED ON TREMONT MICROARCHITECTURE

Intel Atom processors based on the Tremont microarchitecture support MSRs listed in Table 2-6, Table 2-12, Table 2-13 and Table 2-14. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_86H; see Table 2-1. For an MSR listed in Table 2-14 that also appears in the model-specific tables of prior generations, Table 2-14 supersedes prior generation tables.

In the Tremont 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 2-14. MSRs in Intel Atom® Processors Based on the Tremont Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
33H	51	MSR_TEST_CTRL	Core	Test Control Register
		28:0		Reserved.
		29		Enable #AC(0) exception for split locked accesses: Cause #AC(0) exception for split locked access at all CPL irrespective of CR0.AM or EFLAGS.AC. If bits 29 and 31 are both set, bit 29 takes precedence.
		30		Reserved.
		31		Reserved.
CFH	207	IA32_CORE_CAPABILITIES	Core	IA32 Core Capabilities Register If CPUID.(EAX=07H, ECX=0):EDX[30] = 1.
		4:0		Reserved.
		5		Bit 29 of MSR_TEST_CTRL (address 33H) supported.
		63:6		Reserved.
3F1H	1009	MSR_PEBS_ENABLE	Core	(R/W) See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)".
		n:0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMCx. The maximum value n can be determined from CPUID.0AH:EAX[15:8].
		31:n+1		Reserved.
		32+m:32		Enable PEBS trigger and recording for IA32_FIXED_CTRx. The maximum value m can be determined from CPUID.0AH:EDX[4:0].
		59:33+m		Reserved.
		60		Pend a PerfMon Interrupt (PMI) after each PEBS event.
		62:61		Specifies PEBS output destination. Encodings: 00B: DS Save Area 01B: Intel PT trace output. Supported if IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] and CPUID.07H.0.EBX[25] are set. 10B: Reserved 11B: Reserved
		63		Reserved.
1309H - 130BH	4873 - 4875	MSR_RELOAD_FIXED_CTRx		Reload value for IA32_FIXED_CTRx (R/W)
		47:0		Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.
		63:48		Reserved.

Table 2-14. MSRs in Intel Atom® Processors Based on the Tremont Microarchitecture (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
14C1H	5313	MSR_RELOAD_PMCx		Reload value for IA32_PMCx (R/W)
-	-	47:0		Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.
14C8H	5320	63:48		Reserved.

See Table 2-6, Table 2-12, Table 2-13 and Table 2-14 for MSR definitions applicable to processors with CPUID signature 06_86H.

2.8 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 2-15 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 2-16. 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 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.23, "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 17.17, "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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDC-TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDC and TDP Limits for Turbo mode are programmable. When set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, 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, locks bits 15:0 of this register until next reset.
		23:16		Reserved
		24		Interrupt filtering enable (R/W) When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		C3 state auto demotion enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved
186H	390	IA32_PERFEVTSELO	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 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	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 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.
		63:39		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Thread	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees 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)

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control Various model specific features enumeration. See http://biosbits.org .
		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; a value = 1 indicates override is 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; a value = 1 indicates override is 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 17.9.2, "Filtering of Last Branch Records."

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Thread	Provides single-bit status used by software to query the overflow condition of each performance counter. (RO)
		61		UNC_Ovf Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities." Allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Thread	(R/W)

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		61		CLR_UNC_Ovf Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "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
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.3.1.1.2, "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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_MCO_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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_MCO_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 ADDR_V 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 ADDR_V 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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 ADDR_V 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 ADDR_V 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_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."

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_CTL	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_CTL	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_CTL	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_CTL	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_CRO_FIXED0	Thread	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
487H	1159	IA32_VMX_CRO_FIXED1	Thread	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
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."

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTL2	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 18.6.3.4, "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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.9.1 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.
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.

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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)

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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)
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)

Table 2-15. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
C000_0103H		IA32_TSC_AUX	Thread	AUXILIARY TSC Signature (R/W) See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.8.1 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 2-16. 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 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (Not available in model 06_2EH.)

Table 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 2 cores active.
		23:16		Maximum Turbo Ratio Limit 3C (R/O) Maximum Turbo mode ratio limit with 3 cores 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 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
392H	914	MSR_UNCORE_PERF_GLOBAL_STATUS	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
393H	915	MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
394H	916	MSR_UNCORE_FIXED_CTR0	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
395H	917	MSR_UNCORE_FIXED_CTR_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
396H	918	MSR_UNCORE_ADDR_OPCODE_MATCH	Package	See Section 18.3.1.2.3, "Uncore Address/Opcode Match MSR."
3B0H	960	MSR_UNCORE_PMC0	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B1H	961	MSR_UNCORE_PMC1	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B2H	962	MSR_UNCORE_PMC2	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B3H	963	MSR_UNCORE_PMC3	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B4H	964	MSR_UNCORE_PMC4	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

Table 2-16. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3B6H	966	MSR_UNCORE_PMC6	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3B7H	967	MSR_UNCORE_PMC7	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C0H	944	MSR_UNCORE_PERFEVTSELO	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C1H	945	MSR_UNCORE_PERFEVTSEL1	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C2H	946	MSR_UNCORE_PERFEVTSEL2	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C3H	947	MSR_UNCORE_PERFEVTSEL3	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C4H	948	MSR_UNCORE_PERFEVTSEL4	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C5H	949	MSR_UNCORE_PERFEVTSEL5	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C6H	950	MSR_UNCORE_PERFEVTSEL6	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”
3C7H	951	MSR_UNCORE_PERFEVTSEL7	Package	See Section 18.3.1.2.2, “Uncore Performance Event Configuration Facility.”

2.8.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 2-15 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-17. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2EH.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series

Register Address		Register Name / Bit Fields	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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	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.
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.
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."

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	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_SELO	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.
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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	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 1 vperfmon 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_SELO	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.
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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
CBOH	3248	MSR_M0_PMON_EVNT_SEL0	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTRL0	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_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	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_CTRL5	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.
CDOH	3280	MSR_S1_PMON_EVNT_SEL0	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTRL0	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_CTRL1	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_CTRL2	Package	Uncore S-box 1 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D02H	3330	MSR_CO_PMON_BOX_OVF_CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_CO_PMON_EVNT_SELO	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_CO_PMON_CTRL0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_CO_PMON_EVNT_SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_CO_PMON_CTRL1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_CO_PMON_EVNT_SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_CO_PMON_CTRL2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_CO_PMON_EVNT_SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_CO_PMON_CTRL3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_CO_PMON_EVNT_SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_CO_PMON_CTRL4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_CO_PMON_EVNT_SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_CO_PMON_CTRL5	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_SELO	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTRL0	Package	Uncore C-box 4 perfmon counter MSR.
D32H	3378	MSR_C4_PMON_EVNT_SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	Package	Uncore C-box 4 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	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.
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_SELO	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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	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.
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_SELO	Package	Uncore C-box 5 perfmon event select MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DB1H	3505	MSR_C5_PMON_CTRL0	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_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	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_CTRL5	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.
DD0H	3536	MSR_C3_PMON_EVNT_SEL0	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTRL0	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_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	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_CTRL5	Package	Uncore C-box 3 perfmon counter MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTRL0	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_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	Package	Uncore C-box 7 perfmon counter MSR.
DFAH	3578	MSR_C7_PMON_EVNT_SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTRL5	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_IPERFO_P0	Package	Uncore R-box 0 perfmon IPERFO unit Port 0 select MSR.
E05H	3589	MSR_R0_PMON_IPERFO_P1	Package	Uncore R-box 0 perfmon IPERFO unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERFO_P2	Package	Uncore R-box 0 perfmon IPERFO unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERFO_P3	Package	Uncore R-box 0 perfmon IPERFO unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERFO_P4	Package	Uncore R-box 0 perfmon IPERFO unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERFO_P5	Package	Uncore R-box 0 perfmon IPERFO unit Port 5 select MSR.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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 1 perfmon 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.

Table 2-17. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E3EH	3646	MSR_R1_PMON_EVT_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 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

2.9 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 2-15, Table 2-16, plus additional MSR listed in Table 2-18. 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 2-18. Additional MSRs Supported by Intel Processors
(Based on Intel® Microarchitecture Code Name Westmere)**

Register Address		Register Name / Bit Fields	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 instructions 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.

2.10 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 2-15 (except MSR address 1ADH), Table 2-16, plus additional MSR listed in Table 2-19. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2FH.

Table 2-19. Additional MSRs Supported by Intel® Xeon® Processor E7 Family

Register Address		Register Name / Bit Fields	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 instructions 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_SELO	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.
F58H	3928	MSR_C8_PMON_EVNT_SEL4	Package	Uncore C-box 8 perfmon event select MSR.
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR.
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.

Table 2-19. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_SELO	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTRL0	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_CTRL1	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_CTRL2	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_CTRL3	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_CTRL4	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_CTRL5	Package	Uncore C-box 9 perfmon counter MSR.

2.11 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 2-20 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 2-21.

Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.23, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.23, “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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "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 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, 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

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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/W0) When set, locks 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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
		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 instructions 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	Global Machine Check Status
		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_PERFEVTSELO	Thread	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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	Performance Status
		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/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) 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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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 the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.
		63:39		Reserved
1A2H	418	MSR_TEMPERATURE_TARGET	Unique	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees 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).

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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		Miscellaneous Power Management Control Various model specific features enumeration. See http://biosbits.org .
1BOH	432	IA32_ENERGY_PERF_BIAS	Package	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 17.9.2, "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

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
1DDH	477	MSR_LER_FROM_LIP	Thread	Last Exception Record From Linear IP (R/W) 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/W) 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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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	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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	Ovf_PMC0
		1	Thread	Ovf_PMC1
		2	Thread	Ovf_PMC2
		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 18.6.2.2, "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).

Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 18.6.2.2, "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).
		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 18.3.1.1.1, "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)		

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 Section 18.3.1.1.2, "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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_MCO_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS" and Chapter 16.
402H	1026	IA32_MCO_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MCO_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."
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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_CTL	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_CTL	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_CTL	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_CTL	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."
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."

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTL2	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_CTL2	Thread	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL2	Thread	Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTL2	Thread	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTL2	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 18.6.3.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers used in RAPL Interfaces (R/O) See Section 14.10.1, "RAPL Interfaces."
60AH	1546	MSR_PKGC3_IRTL	Package	Package C3 Interrupt Response Limit (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C3 state.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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-state 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 a C6 to a C0 state, where an interrupt request can be delivered to the core and serviced. Additional core-exit latency may 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-state management.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W) See Section 14.10.3, "Package RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.9.1 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.

**Table 2-20. MSRs Supported by Intel® Processors
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-20. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 2-2.
802H-83FH	2050-2111	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 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.11.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-21 and Table 2-22 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 2-21. MSRs Supported by 2nd Generation Intel® Core™ Processors
(Intel® microarchitecture code name Sandy Bridge)**

Register Address		Register Name / Bit Fields	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 a C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may 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-state management.
		63:16		Reserved

Table 2-21. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
63AH	1594	MSR_PP0_POLICY	Package	PP0 Balance Policy (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	PP1 RAPL Power Limit Control (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	PP1 Balance Policy (R/W) See Section 14.10.4, "PP0/PP1 RAPL Domains."

See Table 2-20, Table 2-21, and Table 2-22 for MSR definitions applicable to processors with CPUID signature 06_2AH.

Table 2-22 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06_2AH.

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name / Bit Fields	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 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name / Bit Fields	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_PERFEVTSELO	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_PERFEVTSELO	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_PERFEVTSELO	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
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_PERFEVTSELO	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

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_PERFEVTSELO	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_PERFEVTSELO	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
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

2.11.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-23 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 2-20 and Table 2-24.

Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	MC Bank Error Configuration (R/W)
		0		Reserved

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 cores active.
		23:16	Package	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 cores active.
		31:24	Package	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 cores active.
		39:32	Package	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 cores active.
		47:40	Package	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 cores active.
		55:48	Package	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 cores active.
		63:56	Package	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 cores 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.

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
39CH	924	MSR_PEBS_NUM_ALT	Package	ENABLE_PEBS_NUM_ALT (RW)
		0		ENABLE_PEBS_NUM_ALT (RW) Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see Table 19-19.
		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."
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.

**Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family
(based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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."
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."

Table 2-23. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	PP0 Energy Status (R/O) See Section 14.10.4, "PP0/PP1 RAPL Domains."

See Table 2-20, Table 2-23, and Table 2-24 for MSR definitions applicable to processors with CPUID signature 06_2DH.

2.11.3 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 2-24. 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 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C08H	3080	MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK Fixed Counter Control
C09H	3081	MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK Fixed Counter
C10H	3088	MSR_U_PMON_EVNTSELO	Package	Uncore U-box Perfmon Event Select for U-box Counter 0
C11H	3089	MSR_U_PMON_EVNTSEL1	Package	Uncore U-box Perfmon Event Select for U-box Counter 1
C16H	3094	MSR_U_PMON_CTR0	Package	Uncore U-box Perfmon Counter 0
C17H	3095	MSR_U_PMON_CTR1	Package	Uncore U-box Perfmon Counter 1
C24H	3108	MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU Perfmon for PCU-box-wide Control
C30H	3120	MSR_PCU_PMON_EVNTSELO	Package	Uncore PCU Perfmon Event Select for PCU Counter 0
C31H	3121	MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU Perfmon Event Select for PCU Counter 1
C32H	3122	MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU Perfmon Event Select for PCU Counter 2

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C33H	3123	MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU Perfmon Event Select for PCU Counter 3
C34H	3124	MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU Perfmon box-wide Filter
C36H	3126	MSR_PCU_PMON_CTR0	Package	Uncore PCU Perfmon Counter 0
C37H	3127	MSR_PCU_PMON_CTR1	Package	Uncore PCU Perfmon Counter 1
C38H	3128	MSR_PCU_PMON_CTR2	Package	Uncore PCU Perfmon Counter 2
C39H	3129	MSR_PCU_PMON_CTR3	Package	Uncore PCU Perfmon Counter 3
D04H	3332	MSR_CO_PMON_BOX_CTL	Package	Uncore C-box 0 Perfmon Local Box Wide Control
D10H	3344	MSR_CO_PMON_EVNTSEL0	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 0
D11H	3345	MSR_CO_PMON_EVNTSEL1	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 1
D12H	3346	MSR_CO_PMON_EVNTSEL2	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 2
D13H	3347	MSR_CO_PMON_EVNTSEL3	Package	Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 3
D14H	3348	MSR_CO_PMON_BOX_FILTER	Package	Uncore C-box 0 Perfmon Box Wide Filter
D16H	3350	MSR_CO_PMON_CTR0	Package	Uncore C-box 0 Perfmon Counter 0
D17H	3351	MSR_CO_PMON_CTR1	Package	Uncore C-box 0 Perfmon Counter 1
D18H	3352	MSR_CO_PMON_CTR2	Package	Uncore C-box 0 Perfmon Counter 2
D19H	3353	MSR_CO_PMON_CTR3	Package	Uncore C-box 0 Perfmon Counter 3
D24H	3364	MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 Perfmon Local Box Wide Control
D30H	3376	MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 0
D31H	3377	MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 1
D32H	3378	MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 2
D33H	3379	MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 Perfmon Event Select for C-box 1 Counter 3
D34H	3380	MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 Perfmon Box Wide Filter
D36H	3382	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 Perfmon Counter 0
D37H	3383	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 Perfmon Counter 1
D38H	3384	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 Perfmon Counter 2
D39H	3385	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 Perfmon Counter 3
D44H	3396	MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 Perfmon Local Box Wide Control
D50H	3408	MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 0
D51H	3409	MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 1

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
D52H	3410	MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 2
D53H	3411	MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 3
D54H	3412	MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 Perfmon Box Wide Filter
D56H	3414	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 Perfmon Counter 0
D57H	3415	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 Perfmon Counter 1
D58H	3416	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 Perfmon Counter 2
D59H	3417	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 Perfmon Counter 3
D64H	3428	MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 Perfmon Local Box Wide Control
D70H	3440	MSR_C3_PMON_EVNTSELO	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 0
D71H	3441	MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 1
D72H	3442	MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 2
D73H	3443	MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 3
D74H	3444	MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 Perfmon Box Wide Filter
D76H	3446	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 Perfmon Counter 0
D77H	3447	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 Perfmon Counter 1
D78H	3448	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 Perfmon Counter 2
D79H	3449	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 Perfmon Counter 3
D84H	3460	MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 Perfmon Local Box Wide Control
D90H	3472	MSR_C4_PMON_EVNTSELO	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 0
D91H	3473	MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 1
D92H	3474	MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 2
D93H	3475	MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 3
D94H	3476	MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 Perfmon Box Wide Filter
D96H	3478	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 Perfmon Counter 0
D97H	3479	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 Perfmon Counter 1
D98H	3480	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 Perfmon Counter 2
D99H	3481	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 Perfmon Counter 3
DA4H	3492	MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 Perfmon Local Box Wide Control
DB0H	3504	MSR_C5_PMON_EVNTSELO	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 0

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DB1H	3505	MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 1
DB2H	3506	MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 2
DB3H	3507	MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 3
DB4H	3508	MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 Perfmon Box Wide Filter
DB6H	3510	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 Perfmon Counter 0
DB7H	3511	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 Perfmon Counter 1
DB8H	3512	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 Perfmon Counter 2
DB9H	3513	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 Perfmon Counter 3
DC4H	3524	MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 Perfmon Local Box Wide Control
DD0H	3536	MSR_C6_PMON_EVNTSELO	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 0
DD1H	3537	MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 1
DD2H	3538	MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 2
DD3H	3539	MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 Perfmon Event Select for C-box 6 Counter 3
DD4H	3540	MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 Perfmon Box Wide Filter
DD6H	3542	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 Perfmon Counter 0
DD7H	3543	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 Perfmon Counter 1
DD8H	3544	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 Perfmon Counter 2
DD9H	3545	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 Perfmon Counter 3
DE4H	3556	MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 Perfmon Local Box Wide Control
DF0H	3568	MSR_C7_PMON_EVNTSELO	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 0
DF1H	3569	MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 1
DF2H	3570	MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 2
DF3H	3571	MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 Perfmon Event Select for C-box 7 Counter 3
DF4H	3572	MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 Perfmon Box Wide Filter
DF6H	3574	MSR_C7_PMON_CTR0	Package	Uncore C-box 7 Perfmon Counter 0
DF7H	3575	MSR_C7_PMON_CTR1	Package	Uncore C-box 7 Perfmon Counter 1
DF8H	3576	MSR_C7_PMON_CTR2	Package	Uncore C-box 7 Perfmon Counter 2
DF9H	3577	MSR_C7_PMON_CTR3	Package	Uncore C-box 7 Perfmon Counter 3

2.12 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 2-20, Table 2-21, Table 2-22, and Table 2-25. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3AH.

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates that 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. 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. 03: Reserved
		39:35		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, 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		

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors
(based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	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 .
		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 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, locks 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.		

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:29		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/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
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)

**Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors
(based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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 2-20, Table 2-21 and Table 2-22 for other MSR definitions applicable to processors with CPUID signature 06_3AH.

2.12.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 2-26 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 2-20, and Table 2-26.

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		<p>LockOut (R/WO)</p> <p>If 0, indicates that further writes to MSR_PPIN_CTL is allowed.</p> <p>If 1, indicates that further writes to MSR_PPIN_CTL is disallowed. Writing 1 to this bit is only permitted if the Enable_PPIN bit is clear.</p> <p>The Privileged System Software Inventory Agent should read MSR_PPIN_CTL[bit 1] to determine if MSR_PPIN is accessible.</p> <p>The Privileged System Software Inventory Agent is not expected to write to this MSR.</p>
		1		<p>Enable_PPIN (R/W)</p> <p>If 1, indicates that MSR_PPIN is accessible using RDMSR.</p> <p>If 0, indicates that MSR_PPIN is inaccessible using RDMSR. Any attempt to read MSR_PPIN will cause #GP.</p>
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		<p>Protected Processor Inventory Number (R/O)</p> <p>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'.</p>
CEH	206	MSR_PLATFORM_INFO	Package	<p>Platform Information</p> <p>Contains power management and other model specific features enumeration. See http://biosbits.org.</p>
		7:0		Reserved
		15:8	Package	<p>Maximum Non-Turbo Ratio (R/O)</p> <p>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</p>
		22:16		Reserved
		23	Package	<p>PPIN_CAP (R/O)</p> <p>When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for a privileged system inventory agent to read PPIN from MSR_PPIN.</p> <p>When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.</p>
		27:24		Reserved

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. 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 a temperature offset.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, 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-state 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.

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:11		Reserved
		15		CFG Lock (R/WO) When set, locks 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
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (RO) The minimum temperature at which PROCHOT# will be asserted. The value is degrees 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 if MSR_PLATFORM_INFO.[30] is set.
		63:28		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.

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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.
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 errors from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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." Bank MC11 reports MC errors from a specific channel of the integrated memory controller.
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 errors 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 errors 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	

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
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 errors 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	

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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	
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 errors 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 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."

Table 2-26. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."

See Table 2-20, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06_3EH.

2.12.2 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 2-20, Table 2-26, and Table 2-27.

Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name / Bit Fields	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
		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	Global Machine Check Status (R/WO)
		0		RIPV
		1		EIPV

Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "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)

Table 2-27. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 errors 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 errors 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 errors 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 2-20, Table 2-26 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.

2.12.3 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 2-24 and Table 2-28. 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 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
C00H	3072	MSR_PMON_GLOBAL_CTL	Package	Uncore Perfmon Per-Socket Global Control
C01H	3073	MSR_PMON_GLOBAL_STATUS	Package	Uncore Perfmon Per-Socket Global Status
C06H	3078	MSR_PMON_GLOBAL_CONFIG	Package	Uncore Perfmon Per-Socket Global Configuration
C15H	3093	MSR_U_PMON_BOX_STATUS	Package	Uncore U-box Perfmon U-Box Wide Status
C35H	3125	MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU Perfmon Box Wide Status
D1AH	3354	MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-Box 0 Perfmon Box Wide Filter1
D3AH	3386	MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-Box 1 Perfmon Box Wide Filter1
D5AH	3418	MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-Box 2 Perfmon Box Wide Filter1
D7AH	3450	MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-Box 3 Perfmon Box Wide Filter1
D9AH	3482	MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-Box 4 Perfmon Box Wide Filter1
DBAH	3514	MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-Box 5 Perfmon Box Wide Filter1
DDAH	3546	MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-Box 6 Perfmon Box Wide Filter1
DFAH	3578	MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-Box 7 Perfmon Box Wide Filter1
E04H	3588	MSR_C8_PMON_BOX_CTL	Package	Uncore C-Box 8 Perfmon Local Box Wide Control
E10H	3600	MSR_C8_PMON_EVNTSEL0	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0
E11H	3601	MSR_C8_PMON_EVNTSEL1	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1
E12H	3602	MSR_C8_PMON_EVNTSEL2	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2
E13H	3603	MSR_C8_PMON_EVNTSEL3	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3
E14H	3604	MSR_C8_PMON_BOX_FILTER	Package	Uncore C-Box 8 Perfmon Box Wide Filter
E16H	3606	MSR_C8_PMON_CTR0	Package	Uncore C-Box 8 Perfmon Counter 0
E17H	3607	MSR_C8_PMON_CTR1	Package	Uncore C-Box 8 Perfmon Counter 1
E18H	3608	MSR_C8_PMON_CTR2	Package	Uncore C-Box 8 Perfmon Counter 2
E19H	3609	MSR_C8_PMON_CTR3	Package	Uncore C-Box 8 Perfmon Counter 3
E1AH	3610	MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-Box 8 Perfmon Box Wide Filter1
E24H	3620	MSR_C9_PMON_BOX_CTL	Package	Uncore C-Box 9 Perfmon Local Box Wide Control
E30H	3632	MSR_C9_PMON_EVNTSEL0	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 0
E31H	3633	MSR_C9_PMON_EVNTSEL1	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 1
E32H	3634	MSR_C9_PMON_EVNTSEL2	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 2
E33H	3635	MSR_C9_PMON_EVNTSEL3	Package	Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 3
E34H	3636	MSR_C9_PMON_BOX_FILTER	Package	Uncore C-Box 9 Perfmon Box Wide Filter
E36H	3638	MSR_C9_PMON_CTR0	Package	Uncore C-Box 9 Perfmon Counter 0

Table 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E37H	3639	MSR_C9_PMON_CTR1	Package	Uncore C-Box 9 Perfmon Counter 1
E38H	3640	MSR_C9_PMON_CTR2	Package	Uncore C-Box 9 Perfmon Counter 2
E39H	3641	MSR_C9_PMON_CTR3	Package	Uncore C-Box 9 Perfmon Counter 3
E3AH	3642	MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-Box 9 Perfmon Box Wide Filter1
E44H	3652	MSR_C10_PMON_BOX_CTL	Package	Uncore C-Box 10 Perfmon Local Box Wide Control
E50H	3664	MSR_C10_PMON_EVNTSELO	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 0
E51H	3665	MSR_C10_PMON_EVNTSEL1	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 1
E52H	3666	MSR_C10_PMON_EVNTSEL2	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 2
E53H	3667	MSR_C10_PMON_EVNTSEL3	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 3
E54H	3668	MSR_C10_PMON_BOX_FILTER	Package	Uncore C-Box 10 Perfmon Box Wide Filter
E56H	3670	MSR_C10_PMON_CTR0	Package	Uncore C-Box 10 Perfmon Counter 0
E57H	3671	MSR_C10_PMON_CTR1	Package	Uncore C-Box 10 Perfmon Counter 1
E58H	3672	MSR_C10_PMON_CTR2	Package	Uncore C-Box 10 Perfmon Counter 2
E59H	3673	MSR_C10_PMON_CTR3	Package	Uncore C-Box 10 Perfmon Counter 3
E5AH	3674	MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-Box 10 Perfmon Box Wide Filter1
E64H	3684	MSR_C11_PMON_BOX_CTL	Package	Uncore C-Box 11 Perfmon Local Box Wide Control
E70H	3696	MSR_C11_PMON_EVNTSELO	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 0
E71H	3697	MSR_C11_PMON_EVNTSEL1	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 1
E72H	3698	MSR_C11_PMON_EVNTSEL2	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 2
E73H	3699	MSR_C11_PMON_EVNTSEL3	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 3
E74H	3700	MSR_C11_PMON_BOX_FILTER	Package	Uncore C-Box 11 Perfmon Box Wide Filter
E76H	3702	MSR_C11_PMON_CTR0	Package	Uncore C-Box 11 Perfmon Counter 0
E77H	3703	MSR_C11_PMON_CTR1	Package	Uncore C-Box 11 Perfmon Counter 1
E78H	3704	MSR_C11_PMON_CTR2	Package	Uncore C-Box 11 Perfmon Counter 2
E79H	3705	MSR_C11_PMON_CTR3	Package	Uncore C-Box 11 Perfmon Counter 3
E7AH	3706	MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-Box 11 Perfmon Box Wide Filter1
E84H	3716	MSR_C12_PMON_BOX_CTL	Package	Uncore C-Box 12 Perfmon Local Box Wide Control
E90H	3728	MSR_C12_PMON_EVNTSELO	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0
E91H	3729	MSR_C12_PMON_EVNTSEL1	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1

Table 2-28. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E92H	3730	MSR_C12_PMON_EVNTSEL2	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2
E93H	3731	MSR_C12_PMON_EVNTSEL3	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3
E94H	3732	MSR_C12_PMON_BOX_FILTER	Package	Uncore C-Box 12 Perfmon Box Wide Filter
E96H	3734	MSR_C12_PMON_CTR0	Package	Uncore C-Box 12 Perfmon Counter 0
E97H	3735	MSR_C12_PMON_CTR1	Package	Uncore C-Box 12 Perfmon Counter 1
E98H	3736	MSR_C12_PMON_CTR2	Package	Uncore C-Box 12 Perfmon Counter 2
E99H	3737	MSR_C12_PMON_CTR3	Package	Uncore C-Box 12 Perfmon Counter 3
E9AH	3738	MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-Box 12 Perfmon Box Wide Filter1
EA4H	3748	MSR_C13_PMON_BOX_CTL	Package	Uncore C-Box 13 Perfmon Local Box Wide Control
EBOH	3760	MSR_C13_PMON_EVNTSELO	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0
EB1H	3761	MSR_C13_PMON_EVNTSEL1	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1
EB2H	3762	MSR_C13_PMON_EVNTSEL2	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2
EB3H	3763	MSR_C13_PMON_EVNTSEL3	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3
EB4H	3764	MSR_C13_PMON_BOX_FILTER	Package	Uncore C-Box 13 Perfmon Box Wide Filter
EB6H	3766	MSR_C13_PMON_CTR0	Package	Uncore C-Box 13 Perfmon Counter 0
EB7H	3767	MSR_C13_PMON_CTR1	Package	Uncore C-Box 13 Perfmon Counter 1
EB8H	3768	MSR_C13_PMON_CTR2	Package	Uncore C-Box 13 Perfmon Counter 2
EB9H	3769	MSR_C13_PMON_CTR3	Package	Uncore C-Box 13 Perfmon Counter 3
EBAH	3770	MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-Box 13 Perfmon Box Wide Filter1
EC4H	3780	MSR_C14_PMON_BOX_CTL	Package	Uncore C-Box 14 Perfmon Local Box Wide Control
EDO H	3792	MSR_C14_PMON_EVNTSELO	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0
ED1H	3793	MSR_C14_PMON_EVNTSEL1	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1
ED2H	3794	MSR_C14_PMON_EVNTSEL2	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2
ED3H	3795	MSR_C14_PMON_EVNTSEL3	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3
ED4H	3796	MSR_C14_PMON_BOX_FILTER	Package	Uncore C-Box 14 Perfmon Box Wide Filter
ED6H	3798	MSR_C14_PMON_CTR0	Package	Uncore C-Box 14 Perfmon Counter 0
ED7H	3799	MSR_C14_PMON_CTR1	Package	Uncore C-Box 14 Perfmon Counter 1
ED8H	3800	MSR_C14_PMON_CTR2	Package	Uncore C-Box 14 Perfmon Counter 2
ED9H	3801	MSR_C14_PMON_CTR3	Package	Uncore C-Box 14 Perfmon Counter 3
EDA H	3802	MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-Box 14 Perfmon Box Wide Filter1

2.13 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 2-20, Table 2-21, Table 2-22, and Table 2-29. For an MSR listed in Table 2-20 that also appears in Table 2-29, Table 2-29 supersedes Table 2-20.

The MSRs listed in Table 2-29 also apply to processors based on Haswell-E microarchitecture (see Section 2.14).

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	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	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. 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. 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. 03: Reserved
		39:35		Reserved
	47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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_PERFEVTSELO	Thread	Performance Event Select for Counter 0 (R/W) Supports all fields described in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. 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 in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. 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 in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
		33		IN_TXCP: See Section 18.3.6.5.1. When IN_TXCP=1 & IN_TX=1 and in sampling, a 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 in Table 2-2 and the fields below.
		32		IN_TX: See Section 18.3.6.5.1 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

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	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
		63:9		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
		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_PKG_C7_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 a transition to a 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 a 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.

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12:10		Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 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-state 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 a transition to a 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 a 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 2-20 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-state management.
		63:16		Reserved
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "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).

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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)

Table 2-29. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		7:0		MAX_NON_TURBO_RATIO (R/W/L) System BIOS can program this field.
		30:8		Reserved
		31		TURBO_ACTIVATION_RATIO_Lock (R/W/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.

2.13.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 2-30 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 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture)

Register Address		Register Name / Bit Fields	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 Package C states C7 are not available to processors with a signature of 06_3CH.
		9:4		Reserved
		10		I/O MWAIT Redirection Enable (R/W)

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	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)
		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.
		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

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb Unit, Performance Counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	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		

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
		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.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$; 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 14.10.1, "RAPL Interfaces."
		63:20		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	PP1 RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	PP1 Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	PP1 Balance Policy (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
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 (RO) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		Graphics Driver Status (RO) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		Autonomous Utilization-Based Frequency Control Status (RO) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		VR Therm Alert Status (RO) 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 (RO) 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 (RO) When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		Package-Level Power Limiting PL1 Status (RO) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		Package-Level PL2 Power Limiting Status (RO) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		Max Turbo Limit Status (RO) When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		Turbo Transition Attenuation Status (RO) 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.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17		<p>Thermal Log</p> <p>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		19:18		Reserved
		20		<p>Graphics Driver Log</p> <p>When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		21		<p>Autonomous Utilization-Based Frequency Control Log</p> <p>When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		22		<p>VR Therm Alert Log</p> <p>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		23		Reserved
		24		<p>Electrical Design Point Log</p> <p>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		25		<p>Core Power Limiting Log</p> <p>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>
		26		<p>Package-Level PL1 Power Limiting Log</p> <p>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared.</p> <p>This log bit will remain set until cleared by software writing 0.</p>

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	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.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address		Register Name / Bit Fields	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
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_PERFCTRO	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_PERFCTRO	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_PERFCTRO	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_PERFCTRO	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 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06_46H.				

2.13.2 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 2-20, Table 2-21, Table 2-29, Table 2-30, and Table 2-31.

Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name / Bit Fields	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)
		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

Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
		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 2-20, Table 2-21, Table 2-22, Table 2-29, Table 2-30 for other MSR definitions applicable to processors with CPUID signature 06_45H.

2.14 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 2-20, Table 2-29, and Table 2-32.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	Configured State of Enabled Processor Core Count and Logical Processor Count (RO) <ul style="list-style-type: none"> 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		THREAD_COUNT (RO) The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		31:16		Core_COUNT (RO) The number of processor cores 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
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/W0)
		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		

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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	
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.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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 for 16C 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

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	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 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	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 errors 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 14.10.1, "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 14.10.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "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 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "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 for 100MHz operation (default). 01b: Use 5:4 mapping for 125MHz operation. 10b: Use 5:3 mapping for 166MHz operation. 11b: Use 5:2 mapping for 250MHz operation.
		2	Package	LPLL Select (R/W) If 1, use configured setting of PCIe Ratio.
		3	Package	LONG RESET (R/W) If 1, wait an additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_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.
		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

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
C8DH	3213	IA32_QM_EVTSEL	THREAD	Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.

Table 2-32. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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 2-20, Table 2-29 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.

2.14.1 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 2-33. 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 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
700H	1792	MSR_PMON_GLOBAL_CTL	Package	Uncore Perfmon Per-Socket Global Control
701H	1793	MSR_PMON_GLOBAL_STATUS	Package	Uncore Perfmon Per-Socket Global Status
702H	1794	MSR_PMON_GLOBAL_CONFIG	Package	Uncore Perfmon Per-Socket Global Configuration
703H	1795	MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-Box UCLK Fixed Counter Control
704H	1796	MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-Box UCLK Fixed Counter
705H	1797	MSR_U_PMON_EVNTSELO	Package	Uncore U-Box Perfmon Event Select for U-Box Counter 0

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
706H	1798	MSR_U_PMON_EVNTSEL1	Package	Uncore U-Box Perfmon Event Select for U-Box Counter 1
708H	1800	MSR_U_PMON_BOX_STATUS	Package	Uncore U-Box Perfmon U-Box Wide Status
709H	1801	MSR_U_PMON_CTR0	Package	Uncore U-Box Perfmon Counter 0
70AH	1802	MSR_U_PMON_CTR1	Package	Uncore U-Box Perfmon Counter 1
710H	1808	MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU Perfmon for PCU-Box-Wide Control
711H	1809	MSR_PCU_PMON_EVNTSELO	Package	Uncore PCU Perfmon Event Select for PCU Counter 0
712H	1810	MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU Perfmon Event Select for PCU Counter 1
713H	1811	MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU Perfmon Event Select for PCU Counter 2
714H	1812	MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU Perfmon Event Select for PCU Counter 3
715H	1813	MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU Perfmon Box-Wide Filter
716H	1814	MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU Perfmon Box Wide Status
717H	1815	MSR_PCU_PMON_CTR0	Package	Uncore PCU Perfmon Counter 0
718H	1816	MSR_PCU_PMON_CTR1	Package	Uncore PCU Perfmon Counter 1
719H	1817	MSR_PCU_PMON_CTR2	Package	Uncore PCU Perfmon Counter 2
71AH	1818	MSR_PCU_PMON_CTR3	Package	Uncore PCU Perfmon Counter 3
720H	1824	MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 Perfmon for SBo 0 Box-Wide Control
721H	1825	MSR_S0_PMON_EVNTSELO	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 0
722H	1826	MSR_S0_PMON_EVNTSEL1	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 1
723H	1827	MSR_S0_PMON_EVNTSEL2	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 2
724H	1828	MSR_S0_PMON_EVNTSEL3	Package	Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 3
725H	1829	MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 Perfmon Box-Wide Filter
726H	1830	MSR_S0_PMON_CTR0	Package	Uncore SBo 0 Perfmon Counter 0
727H	1831	MSR_S0_PMON_CTR1	Package	Uncore SBo 0 Perfmon Counter 1
728H	1832	MSR_S0_PMON_CTR2	Package	Uncore SBo 0 Perfmon Counter 2
729H	1833	MSR_S0_PMON_CTR3	Package	Uncore SBo 0 Perfmon Counter 3
72AH	1834	MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 Perfmon for SBo 1 Box-Wide Control
72BH	1835	MSR_S1_PMON_EVNTSELO	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 0
72CH	1836	MSR_S1_PMON_EVNTSEL1	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 1
72DH	1837	MSR_S1_PMON_EVNTSEL2	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 2
72EH	1838	MSR_S1_PMON_EVNTSEL3	Package	Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 3
72FH	1839	MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 Perfmon Box-Wide Filter

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
730H	1840	MSR_S1_PMON_CTR0	Package	Uncore SBo 1 Perfmon Counter 0
731H	1841	MSR_S1_PMON_CTR1	Package	Uncore SBo 1 Perfmon Counter 1
732H	1842	MSR_S1_PMON_CTR2	Package	Uncore SBo 1 Perfmon Counter 2
733H	1843	MSR_S1_PMON_CTR3	Package	Uncore SBo 1 Perfmon Counter 3
734H	1844	MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 Perfmon for SBo 2 Box-Wide Control
735H	1845	MSR_S2_PMON_EVNTSELO	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 0
736H	1846	MSR_S2_PMON_EVNTSEL1	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 1
737H	1847	MSR_S2_PMON_EVNTSEL2	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 2
738H	1848	MSR_S2_PMON_EVNTSEL3	Package	Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 3
739H	1849	MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 Perfmon Box-Wide Filter
73AH	1850	MSR_S2_PMON_CTR0	Package	Uncore SBo 2 Perfmon Counter 0
73BH	1851	MSR_S2_PMON_CTR1	Package	Uncore SBo 2 Perfmon Counter 1
73CH	1852	MSR_S2_PMON_CTR2	Package	Uncore SBo 2 Perfmon Counter 2
73DH	1853	MSR_S2_PMON_CTR3	Package	Uncore SBo 2 Perfmon Counter 3
73EH	1854	MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 Perfmon for SBo 3 Box-Wide Control
73FH	1855	MSR_S3_PMON_EVNTSELO	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 0
740H	1856	MSR_S3_PMON_EVNTSEL1	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 1
741H	1857	MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 2
742H	1858	MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 3
743H	1859	MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 Perfmon Box-Wide Filter
744H	1860	MSR_S3_PMON_CTR0	Package	Uncore SBo 3 Perfmon Counter 0
745H	1861	MSR_S3_PMON_CTR1	Package	Uncore SBo 3 Perfmon Counter 1
746H	1862	MSR_S3_PMON_CTR2	Package	Uncore SBo 3 Perfmon Counter 2
747H	1863	MSR_S3_PMON_CTR3	Package	Uncore SBo 3 Perfmon Counter 3
E00H	3584	MSR_CO_PMON_BOX_CTL	Package	Uncore C-Box 0 Perfmon for Box-Wide Control
E01H	3585	MSR_CO_PMON_EVNTSELO	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 0
E02H	3586	MSR_CO_PMON_EVNTSEL1	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 1
E03H	3587	MSR_CO_PMON_EVNTSEL2	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 2
E04H	3588	MSR_CO_PMON_EVNTSEL3	Package	Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E05H	3589	MSR_CO_PMON_BOX_FILTER0	Package	Uncore C-Box 0 Perfmon Box Wide Filter 0
E06H	3590	MSR_CO_PMON_BOX_FILTER1	Package	Uncore C-Box 0 Perfmon Box Wide Filter 1
E07H	3591	MSR_CO_PMON_BOX_STATUS	Package	Uncore C-Box 0 Perfmon Box Wide Status
E08H	3592	MSR_CO_PMON_CTR0	Package	Uncore C-Box 0 Perfmon Counter 0
E09H	3593	MSR_CO_PMON_CTR1	Package	Uncore C-Box 0 Perfmon Counter 1
E0AH	3594	MSR_CO_PMON_CTR2	Package	Uncore C-Box 0 Perfmon Counter 2
E0BH	3595	MSR_CO_PMON_CTR3	Package	Uncore C-Box 0 Perfmon Counter 3
E10H	3600	MSR_C1_PMON_BOX_CTL	Package	Uncore C-Box 1 Perfmon for Box-Wide Control
E11H	3601	MSR_C1_PMON_EVNTSELO	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 0
E12H	3602	MSR_C1_PMON_EVNTSEL1	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 1
E13H	3603	MSR_C1_PMON_EVNTSEL2	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 2
E14H	3604	MSR_C1_PMON_EVNTSEL3	Package	Uncore C-Box 1 Perfmon Event Select for C-Box 1 Counter 3
E15H	3605	MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-Box 1 Perfmon Box Wide Filter 0
E16H	3606	MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-Box 1 Perfmon Box Wide Filter1
E17H	3607	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-Box 1 Perfmon Box Wide Status
E18H	3608	MSR_C1_PMON_CTR0	Package	Uncore C-Box 1 Perfmon Counter 0
E19H	3609	MSR_C1_PMON_CTR1	Package	Uncore C-Box 1 Perfmon Counter 1
E1AH	3610	MSR_C1_PMON_CTR2	Package	Uncore C-Box 1 Perfmon Counter 2
E1BH	3611	MSR_C1_PMON_CTR3	Package	Uncore C-Box 1 Perfmon Counter 3
E20H	3616	MSR_C2_PMON_BOX_CTL	Package	Uncore C-Box 2 Perfmon for Box-Wide Control
E21H	3617	MSR_C2_PMON_EVNTSELO	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 0
E22H	3618	MSR_C2_PMON_EVNTSEL1	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 1
E23H	3619	MSR_C2_PMON_EVNTSEL2	Package	Uncore C-Box 2 Perfmon Event Select for C-Box 2 Counter 2
E24H	3620	MSR_C2_PMON_EVNTSEL3	Package	Uncore C-Box 2 Perfmon Event select for C-Box 2 Counter 3
E25H	3621	MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-Box 2 Perfmon Box Wide Filter 0
E26H	3622	MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-Box 2 Perfmon Box Wide Filter1
E27H	3623	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-Box 2 Perfmon Box Wide Status
E28H	3624	MSR_C2_PMON_CTR0	Package	Uncore C-Box 2 Perfmon Counter 0
E29H	3625	MSR_C2_PMON_CTR1	Package	Uncore C-Box 2 Perfmon Counter 1
E2AH	3626	MSR_C2_PMON_CTR2	Package	Uncore C-Box 2 Perfmon Counter 2
E2BH	3627	MSR_C2_PMON_CTR3	Package	Uncore C-Box 2 Perfmon Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E30H	3632	MSR_C3_PMON_BOX_CTL	Package	Uncore C-Box 3 Perfmon for Box-Wide Control
E31H	3633	MSR_C3_PMON_EVNTSELO	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 0
E32H	3634	MSR_C3_PMON_EVNTSEL1	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 1
E33H	3635	MSR_C3_PMON_EVNTSEL2	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 2
E34H	3636	MSR_C3_PMON_EVNTSEL3	Package	Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 3
E35H	3637	MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-Box 3 Perfmon Box Wide Filter 0
E36H	3638	MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-Box 3 Perfmon Box Wide Filter1
E37H	3639	MSR_C3_PMON_BOX_STATUS	Package	Uncore C-Box 3 Perfmon Box Wide Status
E38H	3640	MSR_C3_PMON_CTRL0	Package	Uncore C-Box 3 Perfmon Counter 0
E39H	3641	MSR_C3_PMON_CTRL1	Package	Uncore C-Box 3 Perfmon Counter 1
E3AH	3642	MSR_C3_PMON_CTRL2	Package	Uncore C-Box 3 Perfmon Counter 2
E3BH	3643	MSR_C3_PMON_CTRL3	Package	Uncore C-Box 3 Perfmon Counter 3
E40H	3648	MSR_C4_PMON_BOX_CTL	Package	Uncore C-Box 4 Perfmon for Box-Wide Control
E41H	3649	MSR_C4_PMON_EVNTSELO	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 0
E42H	3650	MSR_C4_PMON_EVNTSEL1	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 1
E43H	3651	MSR_C4_PMON_EVNTSEL2	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 2
E44H	3652	MSR_C4_PMON_EVNTSEL3	Package	Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 3
E45H	3653	MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-Box 4 Perfmon Box Wide Filter 0
E46H	3654	MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-Box 4 Perfmon Box Wide Filter1
E47H	3655	MSR_C4_PMON_BOX_STATUS	Package	Uncore C-Box 4 Perfmon Box Wide Status
E48H	3656	MSR_C4_PMON_CTRL0	Package	Uncore C-Box 4 Perfmon Counter 0
E49H	3657	MSR_C4_PMON_CTRL1	Package	Uncore C-Box 4 Perfmon Counter 1
E4AH	3658	MSR_C4_PMON_CTRL2	Package	Uncore C-Box 4 Perfmon Counter 2
E4BH	3659	MSR_C4_PMON_CTRL3	Package	Uncore C-Box 4 Perfmon Counter 3
E50H	3664	MSR_C5_PMON_BOX_CTL	Package	Uncore C-Box 5 Perfmon for Box-Wide Control
E51H	3665	MSR_C5_PMON_EVNTSELO	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 0
E52H	3666	MSR_C5_PMON_EVNTSEL1	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 1
E53H	3667	MSR_C5_PMON_EVNTSEL2	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 2
E54H	3668	MSR_C5_PMON_EVNTSEL3	Package	Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E55H	3669	MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-Box 5 Perfmon Box Wide Filter 0
E56H	3670	MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-Box 5 Perfmon Box Wide Filter 1
E57H	3671	MSR_C5_PMON_BOX_STATUS	Package	Uncore C-Box 5 Perfmon Box Wide Status
E58H	3672	MSR_C5_PMON_CTR0	Package	Uncore C-Box 5 Perfmon Counter 0
E59H	3673	MSR_C5_PMON_CTR1	Package	Uncore C-Box 5 Perfmon Counter 1
E5AH	3674	MSR_C5_PMON_CTR2	Package	Uncore C-Box 5 Perfmon Counter 2
E5BH	3675	MSR_C5_PMON_CTR3	Package	Uncore C-Box 5 Perfmon Counter 3
E60H	3680	MSR_C6_PMON_BOX_CTL	Package	Uncore C-Box 6 Perfmon for Box-Wide Control
E61H	3681	MSR_C6_PMON_EVNTSEL0	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 0
E62H	3682	MSR_C6_PMON_EVNTSEL1	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 1
E63H	3683	MSR_C6_PMON_EVNTSEL2	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 2
E64H	3684	MSR_C6_PMON_EVNTSEL3	Package	Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 3
E65H	3685	MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-Box 6 Perfmon Box Wide Filter 0
E66H	3686	MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-Box 6 Perfmon Box Wide Filter 1
E67H	3687	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-Box 6 Perfmon Box Wide Status
E68H	3688	MSR_C6_PMON_CTR0	Package	Uncore C-Box 6 Perfmon Counter 0
E69H	3689	MSR_C6_PMON_CTR1	Package	Uncore C-Box 6 Perfmon Counter 1
E6AH	3690	MSR_C6_PMON_CTR2	Package	Uncore C-Box 6 Perfmon Counter 2
E6BH	3691	MSR_C6_PMON_CTR3	Package	Uncore C-Box 6 Perfmon Counter 3
E70H	3696	MSR_C7_PMON_BOX_CTL	Package	Uncore C-Box 7 Perfmon for Box-Wide Control
E71H	3697	MSR_C7_PMON_EVNTSEL0	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 0
E72H	3698	MSR_C7_PMON_EVNTSEL1	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 1
E73H	3699	MSR_C7_PMON_EVNTSEL2	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 2
E74H	3700	MSR_C7_PMON_EVNTSEL3	Package	Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 3
E75H	3701	MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-Box 7 Perfmon Box Wide Filter 0
E76H	3702	MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-Box 7 Perfmon Box Wide Filter 1
E77H	3703	MSR_C7_PMON_BOX_STATUS	Package	Uncore C-Box 7 Perfmon Box Wide Status
E78H	3704	MSR_C7_PMON_CTR0	Package	Uncore C-Box 7 Perfmon Counter 0
E79H	3705	MSR_C7_PMON_CTR1	Package	Uncore C-Box 7 Perfmon Counter 1
E7AH	3706	MSR_C7_PMON_CTR2	Package	Uncore C-Box 7 Perfmon Counter 2
E7BH	3707	MSR_C7_PMON_CTR3	Package	Uncore C-Box 7 Perfmon Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
E80H	3712	MSR_C8_PMON_BOX_CTL	Package	Uncore C-Box 8 Perfmon Local Box Wide Control
E81H	3713	MSR_C8_PMON_EVNTSELO	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0
E82H	3714	MSR_C8_PMON_EVNTSEL1	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1
E83H	3715	MSR_C8_PMON_EVNTSEL2	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2
E84H	3716	MSR_C8_PMON_EVNTSEL3	Package	Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3
E85H	3717	MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-Box 8 Perfmon Box Wide Filter 0
E86H	3718	MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-Box 8 Perfmon Box Wide Filter 1
E87H	3719	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-Box 8 Perfmon Box Wide Status
E88H	3720	MSR_C8_PMON_CTRL0	Package	Uncore C-Box 8 Perfmon Counter 0
E89H	3721	MSR_C8_PMON_CTRL1	Package	Uncore C-Box 8 Perfmon Counter 1
E8AH	3722	MSR_C8_PMON_CTRL2	Package	Uncore C-Box 8 Perfmon Counter 2
E8BH	3723	MSR_C8_PMON_CTRL3	Package	Uncore C-Box 8 Perfmon Counter 3
E90H	3728	MSR_C9_PMON_BOX_CTL	Package	Uncore C-Box 9 Perfmon Local Box Wide Control
E91H	3729	MSR_C9_PMON_EVNTSELO	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 0
E92H	3730	MSR_C9_PMON_EVNTSEL1	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 1
E93H	3731	MSR_C9_PMON_EVNTSEL2	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 2
E94H	3732	MSR_C9_PMON_EVNTSEL3	Package	Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 3
E95H	3733	MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-Box 9 Perfmon Box Wide Filter 0
E96H	3734	MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-Box 9 Perfmon Box Wide Filter 1
E97H	3735	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-Box 9 Perfmon Box Wide Status
E98H	3736	MSR_C9_PMON_CTRL0	Package	Uncore C-Box 9 Perfmon Counter 0
E99H	3737	MSR_C9_PMON_CTRL1	Package	Uncore C-Box 9 Perfmon Counter 1
E9AH	3738	MSR_C9_PMON_CTRL2	Package	Uncore C-Box 9 Perfmon Counter 2
E9BH	3739	MSR_C9_PMON_CTRL3	Package	Uncore C-Box 9 Perfmon Counter 3
EA0H	3744	MSR_C10_PMON_BOX_CTL	Package	Uncore C-Box 10 Perfmon Local Box Wide Control
EA1H	3745	MSR_C10_PMON_EVNTSELO	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 0
EA2H	3746	MSR_C10_PMON_EVNTSEL1	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 1
EA3H	3747	MSR_C10_PMON_EVNTSEL2	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 2
EA4H	3748	MSR_C10_PMON_EVNTSEL3	Package	Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
EA5H	3749	MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-Box 10 Perfmon Box Wide Filter 0
EA6H	3750	MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-Box 10 Perfmon Box Wide Filter 1
EA7H	3751	MSR_C10_PMON_BOX_STATUS	Package	Uncore C-Box 10 Perfmon Box Wide Status
EA8H	3752	MSR_C10_PMON_CTR0	Package	Uncore C-Box 10 Perfmon Counter 0
EA9H	3753	MSR_C10_PMON_CTR1	Package	Uncore C-Box 10 Perfmon Counter 1
EA AH	3754	MSR_C10_PMON_CTR2	Package	Uncore C-Box 10 Perfmon Counter 2
EABH	3755	MSR_C10_PMON_CTR3	Package	Uncore C-Box 10 Perfmon Counter 3
EBOH	3760	MSR_C11_PMON_BOX_CTL	Package	Uncore C-Box 11 Perfmon Local Box Wide Control
EB1H	3761	MSR_C11_PMON_EVNTSELO	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 0
EB2H	3762	MSR_C11_PMON_EVNTSEL1	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 1
EB3H	3763	MSR_C11_PMON_EVNTSEL2	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 2
EB4H	3764	MSR_C11_PMON_EVNTSEL3	Package	Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 3
EB5H	3765	MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-Box 11 Perfmon Box Wide Filter 0
EB6H	3766	MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-Box 11 Perfmon Box Wide Filter 1
EB7H	3767	MSR_C11_PMON_BOX_STATUS	Package	Uncore C-Box 11 Perfmon Box Wide Status
EB8H	3768	MSR_C11_PMON_CTR0	Package	Uncore C-Box 11 Perfmon Counter 0
EB9H	3769	MSR_C11_PMON_CTR1	Package	Uncore C-Box 11 Perfmon Counter 1
EBAH	3770	MSR_C11_PMON_CTR2	Package	Uncore C-Box 11 Perfmon Counter 2
EBBH	3771	MSR_C11_PMON_CTR3	Package	Uncore C-Box 11 Perfmon Counter 3
EC0H	3776	MSR_C12_PMON_BOX_CTL	Package	Uncore C-Box 12 Perfmon Local Box Wide Control
EC1H	3777	MSR_C12_PMON_EVNTSELO	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0
EC2H	3778	MSR_C12_PMON_EVNTSEL1	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1
EC3H	3779	MSR_C12_PMON_EVNTSEL2	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2
EC4H	3780	MSR_C12_PMON_EVNTSEL3	Package	Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3
EC5H	3781	MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-Box 12 Perfmon Box Wide Filter 0
EC6H	3782	MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-Box 12 Perfmon Box Wide Filter 1
EC7H	3783	MSR_C12_PMON_BOX_STATUS	Package	Uncore C-Box 12 Perfmon Box Wide Status
EC8H	3784	MSR_C12_PMON_CTR0	Package	Uncore C-Box 12 Perfmon Counter 0
EC9H	3785	MSR_C12_PMON_CTR1	Package	Uncore C-Box 12 Perfmon Counter 1
ECAH	3786	MSR_C12_PMON_CTR2	Package	Uncore C-Box 12 Perfmon Counter 2
ECBH	3787	MSR_C12_PMON_CTR3	Package	Uncore C-Box 12 Perfmon Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
ED0H	3792	MSR_C13_PMON_BOX_CTL	Package	Uncore C-Box 13 Perfmon local box wide control.
ED1H	3793	MSR_C13_PMON_EVNTSELO	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0
ED2H	3794	MSR_C13_PMON_EVNTSEL1	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1
ED3H	3795	MSR_C13_PMON_EVNTSEL2	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2
ED4H	3796	MSR_C13_PMON_EVNTSEL3	Package	Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3
ED5H	3797	MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-Box 13 Perfmon Box Wide Filter 0
ED6H	3798	MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-Box 13 Perfmon Box Wide Filter 1
ED7H	3799	MSR_C13_PMON_BOX_STATUS	Package	Uncore C-Box 13 Perfmon Box Wide Status
ED8H	3800	MSR_C13_PMON_CTR0	Package	Uncore C-Box 13 Perfmon Counter 0
ED9H	3801	MSR_C13_PMON_CTR1	Package	Uncore C-Box 13 Perfmon Counter 1
EDAH	3802	MSR_C13_PMON_CTR2	Package	Uncore C-Box 13 Perfmon Counter 2
EDBH	3803	MSR_C13_PMON_CTR3	Package	Uncore C-Box 13 Perfmon Counter 3
EE0H	3808	MSR_C14_PMON_BOX_CTL	Package	Uncore C-Box 14 Perfmon Local Box Wide Control
EE1H	3809	MSR_C14_PMON_EVNTSELO	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0
EE2H	3810	MSR_C14_PMON_EVNTSEL1	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1
EE3H	3811	MSR_C14_PMON_EVNTSEL2	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2
EE4H	3812	MSR_C14_PMON_EVNTSEL3	Package	Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3
EE5H	3813	MSR_C14_PMON_BOX_FILTER	Package	Uncore C-Box 14 Perfmon Box Wide Filter 0
EE6H	3814	MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-Box 14 Perfmon Box Wide Filter 1
EE7H	3815	MSR_C14_PMON_BOX_STATUS	Package	Uncore C-Box 14 Perfmon Box Wide Status
EE8H	3816	MSR_C14_PMON_CTR0	Package	Uncore C-Box 14 Perfmon Counter 0
EE9H	3817	MSR_C14_PMON_CTR1	Package	Uncore C-Box 14 Perfmon Counter 1
EEAH	3818	MSR_C14_PMON_CTR2	Package	Uncore C-Box 14 Perfmon Counter 2
EEBH	3819	MSR_C14_PMON_CTR3	Package	Uncore C-Box 14 Perfmon Counter 3
EF0H	3824	MSR_C15_PMON_BOX_CTL	Package	Uncore C-Box 15 Perfmon Local Box Wide Control
EF1H	3825	MSR_C15_PMON_EVNTSELO	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 0
EF2H	3826	MSR_C15_PMON_EVNTSEL1	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 1
EF3H	3827	MSR_C15_PMON_EVNTSEL2	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 2
EF4H	3828	MSR_C15_PMON_EVNTSEL3	Package	Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 3

Table 2-33. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
EF5H	3829	MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-Box 15 Perfmon Box Wide Filter 0
EF6H	3830	MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-Box 15 Perfmon Box Wide Filter 1
EF7H	3831	MSR_C15_PMON_BOX_STATUS	Package	Uncore C-Box 15 Perfmon Box Wide Status
EF8H	3832	MSR_C15_PMON_CTR0	Package	Uncore C-Box 15 Perfmon Counter 0
EF9H	3833	MSR_C15_PMON_CTR1	Package	Uncore C-Box 15 Perfmon Counter 1
EFAH	3834	MSR_C15_PMON_CTR2	Package	Uncore C-Box 15 Perfmon Counter 2
EFBH	3835	MSR_C15_PMON_CTR3	Package	Uncore C-Box 15 Perfmon Counter 3
FO0H	3840	MSR_C16_PMON_BOX_CTL	Package	Uncore C-Box 16 Perfmon for Box-Wide Control
F01H	3841	MSR_C16_PMON_EVNTSELO	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 0
F02H	3842	MSR_C16_PMON_EVNTSEL1	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 1
F03H	3843	MSR_C16_PMON_EVNTSEL2	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 2
F04H	3844	MSR_C16_PMON_EVNTSEL3	Package	Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 3
F05H	3845	MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-Box 16 Perfmon Box Wide Filter 0
F06H	3846	MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-Box 16 Perfmon Box Wide Filter 1
F07H	3847	MSR_C16_PMON_BOX_STATUS	Package	Uncore C-Box 16 Perfmon Box Wide Status
F08H	3848	MSR_C16_PMON_CTR0	Package	Uncore C-Box 16 Perfmon Counter 0
F09H	3849	MSR_C16_PMON_CTR1	Package	Uncore C-Box 16 Perfmon Counter 1
F0AH	3850	MSR_C16_PMON_CTR2	Package	Uncore C-Box 16 Perfmon Counter 2
F0BH	3851	MSR_C16_PMON_CTR3	Package	Uncore C-Box 16 Perfmon Counter 3
F10H	3856	MSR_C17_PMON_BOX_CTL	Package	Uncore C-Box 17 Perfmon for Box-Wide Control
F11H	3857	MSR_C17_PMON_EVNTSELO	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 0
F12H	3858	MSR_C17_PMON_EVNTSEL1	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 1
F13H	3859	MSR_C17_PMON_EVNTSEL2	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 2
F14H	3860	MSR_C17_PMON_EVNTSEL3	Package	Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 3
F15H	3861	MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-Box 17 Perfmon Box Wide Filter 0
F16H	3862	MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-Box 17 Perfmon Box Wide Filter1
F17H	3863	MSR_C17_PMON_BOX_STATUS	Package	Uncore C-Box 17 Perfmon Box Wide Status
F18H	3864	MSR_C17_PMON_CTR0	Package	Uncore C-Box 17 Perfmon Counter 0
F19H	3865	MSR_C17_PMON_CTR1	Package	Uncore C-Box 17 Perfmon Counter 1
F1AH	3866	MSR_C17_PMON_CTR2	Package	Uncore C-Box 17 Perfmon Counter 2
F1BH	3867	MSR_C17_PMON_CTR3	Package	Uncore C-Box 17 Perfmon Counter 3

2.15 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 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34, and Table 2-35. For an MSR listed in Table 2-35 that also appears in the model-specific tables of prior generations, Table 2-35 supersedes prior generation tables.

Table 2-34 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 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2. See Section 18.6.2.2, "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 18.6.2.2, "Global Counter Control Facilities."
		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

Table 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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, must be zero.
		2		OS
		3		User
		6:4		Reserved, must be zero.
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		Reserved, must be zero.
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		Reserved; writing 0 will #GP if also setting TraceEn.
		63:14		Reserved, must be zero.
571H	1393	IA32_RTIT_STATUS	Thread	Tracing Status Register (R/W)
		0		Reserved, writes ignored.
		1		ContexEn, writes ignored.
		2		TriggerEn, writes ignored.
		3		Reserved

Table 2-34. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		4		Error (R/W)
		5		Stopped
		63:6		Reserved, must be zero.
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.
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.

NOTES:

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-35 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 2-35. 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 .

Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

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 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		I/O MWAIT Redirection Enable (R/W)
		14:11		Reserved
		15		CFG Lock (R/W0)
		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.

Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."

See Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34 for other MSR definitions applicable to processors with CPUID signature 06_3DH.

2.16 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 2-36 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 2.16.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W/O) See Table 2-26.
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		Protected Processor Inventory Number (R/O) See Table 2-26.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) See Table 2-26.
		22:16		Reserved.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23	Package	PPIN_CAP (R/O) See Table 2-26.
		27:24		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.
		30	Package	Programmable TJ OFFSET (R/O) See Table 2-26.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) See Table 2-26.
		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-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 MWAIT(C1) to MWAIT(C6).
		24:17		Reserved
		25		C3 State Auto Demotion Enable (R/W)

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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/WCO) See Table 2-2.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) See Table 2-2.
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) 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	Temperature Target
		15:0		Reserved

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		23:16		Temperature Target (RO) See Table 2-26.
		27:24		TCC Activation Offset (R/W) See Table 2-26.
		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
		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 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
		15:13	Package	Reserved

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		19:16	Package	Time Units See Section 14.10.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "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 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_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 (R/O) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (R/O) When set, frequency is reduced below the operating system request due to a thermal event.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	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.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	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.
		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

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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.

Table 2-36. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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

2.16.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-37 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 2-20, Table 2-29, Table 2-34, Table 2-36, and Table 2-37.

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name / Bit Fields	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. 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.

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 2-20, Table 2-29, Table 2-34, and Table 2-36 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.

2.16.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-37 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 micro-

architecture and supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-34, Table 2-36, and Table 2-38.

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name / Bit Fields	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.
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 errors 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 errors 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 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name / Bit Fields	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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 2-38. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name / Bit Fields	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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 2-20, Table 2-21, Table 2-29, and Table 2-30 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.

2.17 MSRS IN THE 6TH GENERATION, 7TH GENERATION, 8TH GENERATION, 9TH GENERATION, AND 10TH GENERATION INTEL® CORE™ PROCESSORS, INTEL® XEON® PROCESSOR SCALABLE FAMILY, 8TH GENERATION INTEL® CORE™ I3 PROCESSORS, AND INTEL® XEON® E PROCESSORS

6th generation Intel® Core™ processors and the Intel® Xeon® Processor Scalable Family are based on the Skylake microarchitecture and have CPUID DisplayFamily_DisplayModel signatures of 06_4EH, 06_5EH, and 06_55H.

7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture, 8th generation and 9th generation Intel® Core™ processors and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture; these processors have CPUID DisplayFamily_DisplayModel signatures of 06_8EH and 06_9EH.

8th generation Intel® Core™ i3 processors are based on Cannon Lake microarchitecture and have a CPUID DisplayFamily_DisplayModel signature of 06_66H.

10th generation Intel® Core™ processors are based on Comet Lake microarchitecture (with CPUID DisplayFamily_DisplayModel signatures of 06_A5H, 06_A6H) and Ice Lake microarchitecture (with CPUID DisplayFamily_DisplayModel signatures of 06_7DH and 06_7EH).

These processors support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-25, Table 2-29, Table 2-35, Table 2-39, and Table 2-40. For an MSR listed in Table 2-39 that also appears in the model-specific tables of prior generations, Table 2-39 supersedes 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 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	Control Features in Intel 64 Processor (R/W) See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	MTRR Capability (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/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal threshold #1 Status (RO) See Table 2-2.
		7		Thermal threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
		11		Power Limitation Log (R/WCO) See Table 2-2.
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) 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
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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 avoids 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.
		63:21		Reserved

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
300H	768	MSR_SGXOWNEREPOCH0	Package	Lower 64 Bit CR_SGXOWNEREPOCH (w) 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 (w) 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		

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
390H	912	IA32_PERF_GLOBAL_STATUS_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.
		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).

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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	Thread	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

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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, must be zero.
		7		CR3 filter
		8		ToPA Writing 0 will #GP if also setting TraceEn.
		9		MTCEn
		10		TSCEn
		11		DisRETC
		12		Reserved, must be zero.
		13		BranchEn
		17:14		MTCFreq
		18		Reserved, must be zero.
		22:19		CYCThresh
		23		Reserved, must be zero.
		27:24		PSBFreq
		31:28		Reserved, must be zero.
		35:32		ADDR0_CFG
39:36		ADDR1_CFG		
63:40		Reserved, must be zero.		
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)

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		5		Stopped
		31:6		Reserved, must be zero.
		48:32		PacketByteCnt
		63:49		Reserved, must be zero.
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_ADDRO_A	Thread	Region 0 Start Address (R/W)
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDRO_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_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/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.
		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.Energy_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.)

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		0		PROCHOT Status (RO) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved
		4		Residency State Regulation Status (RO) When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		Running Average Thermal Limit Status (RO) When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		VR Therm Alert Status (RO) 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 (RO) When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		Other Status (RO) 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 (RO) 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 (RO) When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.
		12		Max Turbo Limit Status (RO) When set, frequency is reduced below the operating system request due to multi-core turbo limits.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 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

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_CO	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_CO	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_CO	Package	Any Graphics Engine C0 Residency (R/O)
		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_CO	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*	<p>Platform Power Limit Control (R/W-L)</p> <p>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.</p> <p>The processor implements an exponential-weighted algorithm in the placement of the time windows.</p>

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		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).

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.12.
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 19 From 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.
69AH	1690	MSR_LASTBRANCH_26_FROM_IP	Thread	Last Branch Record 26 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
6BOH	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 (RO) When set, frequency is reduced due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		Running Average Thermal Limit Status (RO) When set, frequency is reduced due to running average thermal limit.
		6		VR Therm Alert Status (RO) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		VR Thermal Design Current Status (RO) When set, frequency is reduced due to VR TDC limit.
		8		Other Status (RO) When set, frequency is reduced due to electrical or other constraints.
		9		Reserved
		10		Package/Platform-Level Power Limiting PL1 Status (RO) When set, frequency is reduced due to package/platform-level power limiting PL1.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		11		Package/Platform-Level PL2 Power Limiting Status (RO) When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.
		12		Inefficient Operation Status (RO) 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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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 (RO) When set, frequency is reduced due to assertion of external PROCHOT.
		1		Thermal Status (RO) When set, frequency is reduced due to a thermal event.
		4:2		Reserved
		5		Running Average Thermal Limit Status (RO) When set, frequency is reduced due to running average thermal limit.
		6		VR Therm Alert Status (RO) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		VR Thermal Design Current Status (RO) When set, frequency is reduced due to VR TDC limit.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
		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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.12.
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 19 To 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 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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".
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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.9.1, "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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-39. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, 8th Generation and 9th Generation Intel® Core™ Processors and Intel® Xeon® E Processors Based on Coffee Lake Microarchitecture, 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture, and 10th Generation Intel® Core™ Processors Based on Comet Lake Microarchitecture and Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
DDFH	3531	MSR_LBR_INFO_31	Thread	Last Branch Record 31 Additional Information (R/W) See description of MSR_LBR_INFO_0.

Table 2-40 lists the MSRs of uncore PMU for Intel processors with CPUID DisplayFamily_DisplayModel signatures of 06_4EH, 06_5EH, 06_8EH, 06_9EH, and 06_66H.

Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors

Register Address		Register Name / Bit Fields	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_PERFEVTSELO	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_PERFEVTSELO	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_PERFEVTSELO	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

Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
716H	1814	MSR_UNC_CBO_1_PERFCTRO	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_PERFEVTSELO	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
726H	1830	MSR_UNC_CBO_2_PERFCTRO	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_PERFEVTSELO	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_PERFCTRO	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		

2.17.1 MSRs Specific to 7th Generation and 8th Generation Intel® Core™ Processors based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Table 2-42 lists additional MSRs for 7th generation and 8th generation Intel Core processors with a CPUID DisplayFamily_DisplayModel signatures of 06_8EH and 06_9EH. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersedes prior generation tables.

Table 2-41. Additional MSRs Supported by 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
80H	128	MSR_TRACE_HUB_STH ACPIBAR_BASE	Package	NPK Address Used by AET Messages (R/W)
		0		Lock Bit If set, this MSR cannot be re-written anymore. Lock bit has to be set in order for the AET packets to be directed to NPK MMIO.
		17:1		Reserved
		63:18		ACPIBAR_BASE_ADDRESS AET target address in NPK MMIO space.
1F4H	500	MSR_PRMRR_PHYS_BASE	Core	Processor Reserved Memory Range Register - Physical Base Control Register (R/W)
		2:0		MemType PRMRR BASE MemType.
		11:3		Reserved
		45:12		Base PRMRR Base Address.
		63:46		Reserved
1F5H	501	MSR_PRMRR_PHYS_MASK	Core	Processor Reserved Memory Range Register - Physical Mask Control Register (R/W)
		9:0		Reserved
		10		Lock Lock bit for the PRMRR.
		11		VLD Enable bit for the PRMRR.
		45:12		Mask PRMRR MASK bits.
		63:46		Reserved
1FBH	507	MSR_PRMRR_VALID_CONFIG	Core	Valid PRMRR Configurations (R/W)
		0		1M supported MEE size.
		4:1		Reserved
		5		32M supported MEE size.
		6		64M supported MEE size.
		7		128M supported MEE size.
		31:8		Reserved

Table 2-41. Additional MSRs Supported by 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
2F4H	756	MSR_UNCORE_PRMR_PHYS_BASE	Package	(R/W) The PRMRR range is used to protect Xucode memory from unauthorized reads and writes. Any IO access to this range is aborted. This register controls the location of the PRMRR range by indicating its starting address. It functions in tandem with the PRMRR mask register.
		11:0		Reserved
		38:12		Range Base This field corresponds to bits 38:12 of the base address memory range which is allocated to PRMRR memory.
		63:39		Reserved
2F5H	757	MSR_UNCORE_PRMR_PHYS_MASK	Package	(R/W) This register controls the size of the PRMRR range by indicating which address bits must match the PRMRR base register value.
		9:0		Reserved
		10		Lock Setting this bit locks all writeable settings in this register, including itself.
		11		Range_En Indicates whether the PRMRR range is enabled and valid.
		38:12		Range_Mask This field indicates which address bits must match PRMRR base in order to qualify as an PRMRR access.
		63:39		Reserved
620H	1568	MSR_RING_RATIO_LIMIT	Package	Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.
		6:0		MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.
		7		Reserved
		14:8		MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.
		63:15		Reserved

2.17.2 MSRs Specific to 8th Generation Intel® Core™ i3 Processors

Table 2-42 lists additional MSRs for 8th generation Intel Core i3 processors with a CPUID DisplayFamily_DisplayModel signature of 06_66H. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersedes prior generation tables.

Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name / Bit Fields	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)
		17		SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Available only if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		SGX Global Functions Enable (R/WL)
		63:21		Reserved
350H	848	MSR_BR_DETECT_CTRL		Branch Monitoring Global Control (R/W)
		0		EnMonitoring Global enable for branch monitoring.
		1		EnExcept Enable branch monitoring event signaling on threshold trip. The branch monitoring event handler is signaled via the existing PMI signaling mechanism as programmed from the corresponding local APIC LVT entry.
		2		EnLBRFrz Enable LBR freeze on threshold trip. This will cause the LBR frozen bit 58 to be set in IA32_PERF_GLOBAL_STATUS when a triggering condition occurs and this bit is enabled.
		3		DisableInGuest When set to '1', branch monitoring, event triggering and LBR freeze actions are disabled when operating at VMX non-root operation.
		7:4		Reserved

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		17:8		WindowSize Window size defined by WindowCntSel. Values 0 - 1023 are supported. Once the Window counter reaches the WindowSize count both the Window Counter and all Branch Monitoring Counters are cleared.
		23:18		Reserved
		25:24		WindowCntSel Window event count select: '00 = Instructions retired. '01 = Branch instructions retired '10 = Return instructions retired. '11 = Indirect branch instructions retired.
		26		CntAndMode When set to '1', the overall branch monitoring event triggering condition is true only if all enabled counters' threshold conditions are true. When '0', the threshold tripping condition is true if any enabled counters' threshold is true.
		63:27		Reserved
351H	849	MSR_BR_DETECT_STATUS		Branch Monitoring Global Status (R/W)
		0		Branch Monitoring Event Signaled When set to '1', Branch Monitoring event signaling is blocked until this bit is cleared by software.
		1		LBRsValid This status bit is set to '1' if the LBR state is considered valid for sampling by branch monitoring software.
		7:2		Reserved
		8		CntrHit0 Branch monitoring counter #0 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		9		CntrHit1 Branch monitoring counter #1 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		15:10		Reserved Reserved for additional branch monitoring counters threshold hit status.

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		25:16		CountWindow The current value of the window counter. The count value is frozen on a valid branch monitoring triggering condition. This is a 10-bit unsigned value.
		31:26		Reserved Reserved for future extension of CountWindow.
		39:32		Count0 The current value of counter 0 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit0 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		47:40		Count1 The current value of counter 1 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit1 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		63:48		Reserved
354H - 355H	852 - 853	MSR_BR_DETECT_COUNTER_CONFIG_i		Branch Monitoring Detect Counter Configuration (R/W)
		0		CntrEn Enable counter.
		7:1		CntrEvSel Event select (other values #GP) '0000000 = RETs. '0000001 = RET-CALL bias. '0000010 = RET mispredicts. '0000011 = Branch (all) mispredicts. '0000100 = Indirect branch mispredicts. '0000101 = Far branch instructions.

**Table 2-42. Additional MSRs Supported by 8th Generation Intel® Core™ i3 Processors
Based on Cannon Lake Microarchitecture (Contd.)**

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		14:8		CntrThreshold Threshold (an unsigned value of 0 to 127 supported). The value 0 of counter threshold will result in event signaled after every instruction. #GP if threshold is < 2.
		15		MispredEventCnt Mispredict events counting behavior: '0 = Mispredict events are counted in a window. '1 = Mispredict events are counted based on a consecutive occurrence. CntrThreshold is treated as # of consecutive mispredicts. This control bit only applies to events specified by CntrEvSel that involve a prediction (0000010, 0000011, 0000100). Setting this bit for other events is ignored.
		63:16		Reserved
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Package C3 Residency Counter (R/O)
		63:0		Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
620H	1568	MSR_RING_RATIO_LIMIT	Package	Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.
		6:0		MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.
		7		Reserved
		14:8		MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.
		63:15		Reserved
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Core C1 Residency Counter (R/O)
		63:0		Value since last reset for the Core C1 residency. Counter rate is the Max Non-Turbo frequency (same as TSC). This counter counts in case both of the core's threads are in an idle state and at least one of the core's thread residency is in a C1 state or in one of its sub states. The counter is updated only after a core C state exit. Note: Always reads 0 if core C1 is unsupported. A value of zero indicates that this processor does not support core C1 or never entered core C1 level state.
662H	1634	MSR_CORE_C3_RESIDENCY	Core	Core C3 Residency Counter (R/O)
		63:0		Will always return 0.

Table 2-43 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06_66H.

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name / Bit Fields	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
		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_PERFCTRO	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_PERFEVTSELO	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_PERFEVTSELO	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_PERFCTRO	Package	Uncore C-Box 0, Performance Counter 0
703H	1795	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, Performance Counter 1
708H	1800	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, Counter 0 Event Select MSR
709H	1801	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, Counter 1 Event Select MSR
70AH	1802	MSR_UNC_CBO_1_PERFCTRO	Package	Uncore C-Box 1, Performance Counter 0
70BH	1803	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, Performance Counter 1
710H	1808	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, Counter 0 Event Select MSR
711H	1809	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, Counter 1 Event Select MSR
712H	1810	MSR_UNC_CBO_2_PERFCTRO	Package	Uncore C-Box 2, Performance Counter 0
713H	1811	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, Performance Counter 1
718H	1816	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, Counter 0 Event Select MSR
719H	1817	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, Counter 1 Event Select MSR
71AH	1818	MSR_UNC_CBO_3_PERFCTRO	Package	Uncore C-Box 3, Performance Counter 0
71BH	1819	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, Performance Counter 1

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
720H	1824	MSR_UNC_CBO_4_PERFEVTSELO	Package	Uncore C-Box 4, Counter 0 Event Select MSR
721H	1825	MSR_UNC_CBO_4_PERFEVTSEL1	Package	Uncore C-Box 4, Counter 1 Event Select MSR
722H	1826	MSR_UNC_CBO_4_PERFCTRO	Package	Uncore C-Box 4, Performance Counter 0
723H	1827	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, Performance Counter 1
728H	1832	MSR_UNC_CBO_5_PERFEVTSELO	Package	Uncore C-Box 5, Counter 0 Event Select MSR
729H	1833	MSR_UNC_CBO_5_PERFEVTSEL1	Package	Uncore C-Box 5, Counter 1 Event Select MSR
72AH	1834	MSR_UNC_CBO_5_PERFCTRO	Package	Uncore C-Box 5, Performance Counter 0
72BH	1835	MSR_UNC_CBO_5_PERFCTR1	Package	Uncore C-Box 5, Performance Counter 1
730H	1840	MSR_UNC_CBO_6_PERFEVTSELO	Package	Uncore C-Box 6, Counter 0 Event Select MSR
731H	1841	MSR_UNC_CBO_6_PERFEVTSEL1	Package	Uncore C-Box 6, Counter 1 Event Select MSR
732H	1842	MSR_UNC_CBO_6_PERFCTRO	Package	Uncore C-Box 6, Performance Counter 0
733H	1843	MSR_UNC_CBO_6_PERFCTR1	Package	Uncore C-Box 6, Performance Counter 1
738H	1848	MSR_UNC_CBO_7_PERFEVTSELO	Package	Uncore C-Box 7, Counter 0 Event Select MSR
739H	1849	MSR_UNC_CBO_7_PERFEVTSEL1	Package	Uncore C-Box 7, Counter 1 Event Select MSR
73AH	1850	MSR_UNC_CBO_7_PERFCTRO	Package	Uncore C-Box 7, Performance Counter 0
73BH	1851	MSR_UNC_CBO_7_PERFCTR1	Package	Uncore C-Box 7, 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	

2.17.3 MSRs Specific to 10th Generation Intel® Core™ Processors

Table 2-44 lists additional MSRs for 10th generation Intel Core processors with a CPUID DisplayFamily_DisplayModel signature values of 06_7DH and 06_7EH. For an MSR listed in Table 2-44 that also appears in the model-specific tables of prior generations, Table 2-44 supersedes prior generation tables.

Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
33H	51	MSR_TEST_CTRL	Core	Test Control Register
		28:0		Reserved.
		29		Enable #AC(0) exception for split locked accesses: Cause #AC(0) exception for split locked access at all CPL irrespective of CRO.AM or EFLAGS.AC. If bits 29 and 31 are both set, bit 29 takes precedence.
		30		Reserved.
		31		Reserved.
48H	72	IA32_SPEC_CTRL	Core	See Table 2-2.
49H	73	IA32_PREDICT_CMD	Thread	See Table 2-2.
8CH	140	IA32_SGXLEPUBKEYHASH0	Thread	See Table 2-2.
8DH	141	IA32_SGXLEPUBKEYHASH1	Thread	See Table 2-2.
8EH	142	IA32_SGXLEPUBKEYHASH2	Thread	See Table 2-2.
8FH	143	IA32_SGXLEPUBKEYHASH3	Thread	See Table 2-2.
A0H	160	MSR_BIOS_MCU_ERRORCODE	Package	BIOS MCU ERRORCODE (R/O) This MSR indicates if WRMSR 0x79 failed to configure PRM memory and gives a hint to debug BIOS.
		15:0	Package	Error Codes (R/O)
		30:16		Reserved.
		31	Thread	MCU Partial Success (R/O) When set to 1, WRMSR 0x79 skipped part of the functionality during BIOS.
A5H	165	MSR_FIT_BIOS_ERROR	Thread	FIT BIOS ERROR (R/W) Report error codes for debug in case the processor failed to parse the Firmware Table in BIOS. Can also be used to log BIOS information.
		7:0		Error Codes (R/W) Error codes for debug.
		15:8		Entry Type (R/W) Failed FIT entry type.
		16		FIT MCU Entry (R/W) FIT contains MCU entry.
		62:17		Reserved.
		63		LOCK (R/W) When set to 1, writes to this MSR will be skipped.

Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
10BH	267	IA32_FLUSH_CMD	Thread	See Table 2-2.
151H	337	MSR_BIOS_DONE	Thread	BIOS Done (R/WO)
		0	Thread	BIOS Done Indication (R/WO) Set by BIOS when it finishes programming the processor and wants to lock the memory configuration from changes by software that is running on this thread. Writes to the bit will be ignored if EAX[0] is 0.
		1	Package	Package BIOS Done Indication (R/O) When set to 1, all threads in the package have bit 0 of this MSR set.
		31:2		Reserved.
1F1H	497	MSR_CRASHLOG_CONTROL	Thread	Write Data to a Crash Log Configuration
		0		CDDIS: CrashDump_Disable If set, indicates that Crash Dump is disabled.
		63:1		Reserved.
2A0H	672	MSR_PRMRR_BASE_0	Core	Processor Reserved Memory Range Register - Physical Base Control Register (R/W)
		2:0		MEMTYPE: PRMRR BASE Memory Type.
		3		CONFIGURED: PRMRR BASE Configured.
		11:4		Reserved.
		51:12		BASE: PRMRR Base Address.
		63:52		Reserved.
30CH	780	IA32_FIXED_CTR3	Thread	Fixed-Function Performance Counter Register 3 (R/W) Bit definitions are the same as found in IA32_FIXED_CTR0, offset 309H. See Table 2-2.
329H	809	MSR_PERF_METRICS	Thread	Performance Metrics (R/W) Reports metrics directly. Software can check (and/or expose to its guests) the availability of PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15).
		7:0		Retiring. Percent of utilized slots by uops that eventually retire (commit).
		15:8		Bad Speculation. Percent of wasted slots due to incorrect speculation, covering utilized by uops that do not retire, or recovery bubbles (unutilized slots).
		23:16		Frontend Bound. Percent of unutilized slots where front-end did not deliver a uop while back-end is ready.
		31:24		Backend Bound. Percent of unutilized slots where a uop was not delivered to back-end due to lack of back-end resources.

Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		63:25		Reserved.
3F2H	1010	MSR_PEBS_DATA_CFG	Thread	PEBS Data Configuration (R/W) Provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.
		0		Memory Info. Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.
		1		GPRs. Setting this bit will capture the contents of the General Purpose registers in the PEBS record.
		2		XMMs. Setting this bit will capture the contents of the XMM registers in the PEBS record.
		3		LBRs. Setting this bit will capture LBR TO, FROM and INFO in the PEBS record.
		23:4		Reserved.
		31:24		LBR Entries. Set the field to the desired number of entries - 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, software can use LBR_entries that is multiple of 3.
541H	1345	MSR_CORE_UARCH_CTL	Core	Core Microarchitecture Control MSR (R/W)
		0		L1 Scrubbing Enable When set to 1, enable L1 scrubbing.
		31:1		Reserved.
657H	1623	MSR_FAST_UNCORE_MSRS_CTL	Thread	Fast WRMSR/RDMSR Control MSR (R/W)
		3:0		FAST_ACCESS_ENABLE: Bit 0: When set to '1', provides a hint for the hardware to enable fast access mode for the IA32_HWP_REQUEST MSR. This bit is sticky and is cleaned by the hardware only during reset time. This bit is valid only if FAST_UNCORE_MSRS_CAPABILITY[0] is set. Setting this bit will cause CPUID[6].EAX[18] to be set.
		31:4		Reserved.

Table 2-44. MSRs Supported by 10th Generation Intel® Core™ Processors Based on Ice Lake Microarchitecture

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
65EH	1630	MSR_FAST_UNCORE_MSRS_STATUS	Thread	Indication of Uncore MSRs, Post Write Activates
		0		Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor. Software can use the status of this bit to avoid overwriting IA32_HWP_REQUEST.
		31:1		Reserved.
65FH	1631	MSR_FAST_UNCORE_MSRS_CAPABILITY	Thread	Fast WRMSR/RDMSR Enumeration MSR (RO)
		3:0		MSRS_CAPABILITY: Bit 0: If set to '1', hardware supports the fast access mode for the IA32_HWP_REQUEST MSR.
		31:4		Reserved.
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Table 2-2.
775H	1909	IA32_PECI_HWP_REQUEST_INFO	Thread	See Table 2-2.
777H	1911	IA32_HWP_STATUS	Thread	See Table 2-2.

2.17.4 MSRs Specific to Intel® Xeon® Processor Scalable Family

Intel® Xeon® Processor Scalable Family (CPUID DisplayFamily_DisplayModel = 06_55H) support the MSRs listed in Table 2-45.

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	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_ENABLED (R/WL)
		63:21		Reserved

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W0) See Table 2-26.
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		Protected Processor Inventory Number (R/O) See Table 2-26.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) See Table 2-26.
		22:16		Reserved.
		23	Package	PPIN_CAP (R/O) See Table 2-26.
		27:24		Reserved
		28	Package	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.
		30	Package	Programmable TJ OFFSET (R/O) See Table 2-26.
		39:31		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) See Table 2-26.
		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 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	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 MWait(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 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	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 is 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 is 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/WCO) See Table 2-2.
		2		PROTCHOT # or FORCEPR# Status (RO) See Table 2-2.
		3		PROTCHOT # or FORCEPR# Log (R/WCO) See Table 2-2.
		4		Critical Temperature Status (RO) See Table 2-2.
		5		Critical Temperature Status Log (R/WCO) See Table 2-2.
		6		Thermal Threshold #1 Status (RO) See Table 2-2.
		7		Thermal Threshold #1 Log (R/WCO) See Table 2-2.
		8		Thermal Threshold #2 Status (RO) See Table 2-2.
		9		Thermal Threshold #2 Log (R/WCO) See Table 2-2.
		10		Power Limitation Status (RO) See Table 2-2.
11		Power Limitation Log (R/WCO) See Table 2-2.		

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		12		Current Limit Status (RO) See Table 2-2.
		13		Current Limit Log (R/WCO) See Table 2-2.
		14		Cross Domain Limit Status (RO) See Table 2-2.
		15		Cross Domain Limit Log (R/WCO) 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	Temperature Target
		15:0		Reserved
		23:16		Temperature Target (RO) See Table 2-26.
		27:24		TCC Activation Offset (R/W) See Table 2-26.
		63:28		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	This register defines the ratio limits. RATIO[0:7] must be populated in ascending order. RATIO[i+1] must be less than or equal to RATIO[i]. Entries with RATIO[i] will be ignored. If any of the rules above are broken, the configuration is silently rejected. If the programmed ratio is: <ul style="list-style-type: none"> ▪ Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !IOC). ▪ Below the min supported ratio, it will be clipped.
		7:0		RATIO_0 Defines ratio limits.
		15:8		RATIO_1 Defines ratio limits.
		23:16		RATIO_2 Defines ratio limits.
		31:24		RATIO_3 Defines ratio limits.

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		39:32		RATIO_4 Defines ratio limits.
		47:40		RATIO_5 Defines ratio limits.
		55:48		RATIO_6 Defines ratio limits.
		63:56		RATIO_7 Defines ratio limits.
1AEH	430	MSR_TURBO_RATIO_LIMIT_CORES	Package	This register defines the active core ranges for each frequency point. NUMCORE[0:7] must be populated in ascending order. NUMCORE[i+1] must be greater than NUMCORE[i]. Entries with NUMCORE[i] == 0 will be ignored. The last valid entry must have NUMCORE >= the number of cores in the SKU. If any of the rules above are broken, the configuration is silently rejected.
		7:0		NUMCORE_0 Defines the active core ranges for each frequency point.
		15:8		NUMCORE_1 Defines the active core ranges for each frequency point.
		23:16		NUMCORE_2 Defines the active core ranges for each frequency point.
		31:24		NUMCORE_3 Defines the active core ranges for each frequency point.
		39:32		NUMCORE_4 Defines the active core ranges for each frequency point.
		47:40		NUMCORE_5 Defines the active core ranges for each frequency point.
		55:48		NUMCORE_6 Defines the active core ranges for each frequency point.
		63:56		NUMCORE_7 Defines the active core ranges for each frequency point.
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.

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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_MCO_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs" through Section 15.3.2.4, "IA32_MCi_MISC MSRs". Bank MCO reports MC errors from the IFU module.
401H	1025	IA32_MCO_STATUS	Core	
402H	1026	IA32_MCO_ADDR	Core	
403H	1027	IA32_MCO_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 errors 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 errors 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 errors from the MLC module.
40DH	1037	IA32_MC3_STATUS	Core	
40EH	1038	IA32_MC3_ADDR	Core	
40FH	1039	IA32_MC3_MISC	Core	
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 errors from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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 errors 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	
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 errors from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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 errors 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 errors 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 errors 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 errors 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 errors 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 errors from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	Unit Multipliers Used in RAPL Interfaces (R/O)
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$; 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 14.10.1, "RAPL Interfaces."
		63:20		Reserved

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "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 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
620H	1568	MSR_UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	Reserved (R/O) Reads return 0.
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

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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.

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
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.

Table 2-45. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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

2.18 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND INTEL® XEON PHI™ PROCESSOR 7215/7285/7295 SERIES

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily_DisplayModel signature 06_57H, supports the MSR interfaces listed in Table 2-46. These processors are based on the Knights Landing microarchitecture. Intel® Xeon Phi™ processor 7215, 7285, 7295 series, with CPUID DisplayFamily_DisplayModel signature 06_85H, supports the MSR interfaces listed in Table 2-46 and Table 2-47. These processors are based on the Knights Mill microarchitecture. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.23, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." See Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "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)

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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/W/O) See Table 2-26.
		1		Enable_PPIN (R/W) See Table 2-26.
		63:2		Reserved
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		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	Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org .
		7:0		Reserved
		15:8	Package	Maximum Non-Turbo Ratio (R/O) This 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 Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.
		29	Package	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		39:30		Reserved
		47:40	Package	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Package	C-State Configuration Control (R/W)
		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, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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)
		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, the AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions 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

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
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_PERFEVTSELO	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

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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.
19CH	412	IA32_THERM_STATUS	Module	Thermal Monitor Status (R/W) See Table 2-2.
		0		Thermal Status (RO)
		1		Thermal Status Log (R/WCO)
		2		PROTCHOT # or FORCEPR# Status (RO)
		3		PROTCHOT # or FORCEPR# Log (R/WCO)
		4		Critical Temperature Status (RO)
		5		Critical Temperature Status Log (R/WCO)
		6		Thermal Threshold #1 Status (RO)
		7		Thermal Threshold #1 Log (R/WCO)
		8		Thermal Threshold #2 Status (RO)
		9		Thermal Threshold #2 Log (R/WCO)
		10		Power Limitation Status (RO)
		11		Power Limitation Log (RWCO)
		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

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		34		XD Bit Disable (R/W)
		37:35		Reserved
		38		Turbo Mode Disable (R/W)
63:39		Reserved		
1A2H	418	MSR_TEMPERATURE_TARGET	Package	Temperature Target
		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.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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".
		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.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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 17.9.2, "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 0-2) 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.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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
		14		FREEZE_WHILE_SMM 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.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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	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	

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		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."
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 ADDR_V 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)

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
		3:0	Package	Power Units See Section 14.10.1, "RAPL Interfaces."
		7:4	Package	Reserved
		12:8	Package	Energy Status Units Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$; 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 14.10.1, "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 14.10.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	PKG Energy Status (R/O) See Section 14.10.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	PKG Perf Status (R/O) See Section 14.10.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	PKG RAPL Parameters (R/W) See Section 14.10.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	DRAM RAPL Power Limit Control (R/W) See Section 14.10.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	DRAM Energy Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	DRAM Performance Throttling Status (R/O) See Section 14.10.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	DRAM RAPL Parameters (R/W) See Section 14.10.5, "DRAM RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	PPO RAPL Power Limit Control (R/W) See Section 14.10.4, "PPO/PP1 RAPL Domains."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	PPO Energy Status (R/O) See Section 14.10.4, "PPO/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	Base TDP Ratio (R/O) See Table 2-25.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O) See Table 2-25.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O) See Table 2-25.
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	ConfigTDP Control (R/W) See Table 2-25.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	ConfigTDP Control (R/W) See Table 2-25.
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)
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_TMRO	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)

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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)
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.

Table 2-46. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
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

Table 2-47 lists model-specific registers that are supported by Intel® Xeon Phi™ processor 7215, 7285, 7295 series based on the Knights Mill microarchitecture.

Table 2-47. Additional MSRs Supported by Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
9BH	155	IA32_SMM_MONITOR_CTL	Core	SMM Monitor Configuration (R/W) This MSR is readable only if VMX is enabled, and writeable only if VMX is enabled and in SMM mode, and is used to configure the VMX MSEG base address. See Table 2-2.
480H	1152	IA32_VMX_BASIC	Core	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2.
481H	1153	IA32_VMX_PINBASED_CTL	Core	Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.
482H	1154	IA32_VMX_PROCBASED_CTL	Core	Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)
483H	1155	IA32_VMX_EXIT_CTL	Core	Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTL	Core	Capability Reporting Register of VM-entry Controls (R/O) See Table 2-2.
485H	1157	IA32_VMX_MISC	Core	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2.
486H	1158	IA32_VMX_CRO_FIXED0	Core	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Table 2-2.
487H	1159	IA32_VMX_CRO_FIXED1	Core	Capability Reporting Register of CRO Bits Fixed to 1 (R/O) See Table 2-2.

Table 2-47. Additional MSRs Supported by Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name / Bit Fields	Scope	Bit Description
Hex	Dec			
488H	1160	IA32_VMX_CR4_FIXED0	Core	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2.
489H	1161	IA32_VMX_CR4_FIXED1	Core	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	Core	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2.
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Core	Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Table 2-2.
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_CTLSS	Core	Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2.
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLSS	Core	Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2.
48FH	1167	IA32_VMX_TRUE_EXIT_CTLSS	Core	Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2.
490H	1168	IA32_VMX_TRUE_ENTRY_CTLSS	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.

2.19 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-48 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 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique ¹	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 2.23, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 2.23, "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."
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.

Table 2-48. 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				
		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.
		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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
		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.
		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: <u>Encoding Scalable Bus Speed</u> 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

Table 2-48. 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				
		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
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: <u>Encoding Scalable Bus Speed</u> 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."

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
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."
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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
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."
		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	395-399	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
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."
		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."

Table 2-48. 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.
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 14.8.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	Thermal Interrupt Control (R/W) See Section 14.8.2, "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 14.8.2, "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)

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
		0			Fast-Strings Enable. See Table 2-2.
		1			Reserved
		2			x87 FPU Fopcode Compatibility Mode Enable
		3			Thermal Monitor 1 Enable See Section 14.8.2, "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 CRO, 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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
					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.
					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 CUID 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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
		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
		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

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
		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 17.13.3, "Last Exception Records."
		31:0			From Linear IP Linear address of the last branch instruction.
		63:32			Reserved
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 17.13.3, "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 17.13.1, "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 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture"; and addresses 1DBH-1DEH and 680H-68FH.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	<p>Last Branch Record 0 (R/O)</p> <p>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.</p> <p>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.</p>
					See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
1DCH	477	MSR_LASTBRANCH_1	0, 1, 2	Unique	<p>Last Branch Record 1</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	<p>Last Branch Record 2</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	<p>Last Branch Record 3</p> <p>See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Base MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	<p>Variable Range Mask MTRR</p> <p>See Section 11.11.2.3, "Variable Range MTRRs."</p>

Table 2-48. 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				
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."
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."

Table 2-48. 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				
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
301H	769	MSR_BPU_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
302H	770	MSR_BPU_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
303H	771	MSR_BPU_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
304H	772	MSR_MS_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
305H	773	MSR_MS_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
306H	774	MSR_MS_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
307H	775	MSR_MS_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
308H	776	MSR_FLAME_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
309H	777	MSR_FLAME_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30AH	778	MSR_FLAME_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30BH	779	MSR_FLAME_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30CH	780	MSR_IQ_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30DH	781	MSR_IQ_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30EH	782	MSR_IQ_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30FH	783	MSR_IQ_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
310H	784	MSR_IQ_COUNTER4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
311H	785	MSR_IQ_COUNTER5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
360H	864	MSR_BPU_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
361H	865	MSR_BPU_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
362H	866	MSR_BPU_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
363H	867	MSR_BPU_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."

Table 2-48. 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				
364H	868	MSR_MS_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
365H	869	MSR_MS_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
366H	870	MSR_MS_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
367H	871	MSR_MS_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
368H	872	MSR_FLAME_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
369H	873	MSR_FLAME_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36AH	874	MSR_FLAME_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36BH	875	MSR_FLAME_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36CH	876	MSR_IQ_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36DH	877	MSR_IQ_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36EH	878	MSR_IQ_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36FH	879	MSR_IQ_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
370H	880	MSR_IQ_CCCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
371H	881	MSR_IQ_CCCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
3A0H	928	MSR_BSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A1H	929	MSR_BSU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A2H	930	MSR_FSB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A3H	931	MSR_FSB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A4H	932	MSR_FIRM_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A5H	933	MSR_FIRM_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A6H	934	MSR_FLAME_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A7H	935	MSR_FLAME_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
3A8H	936	MSR_DAC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A9H	937	MSR_DAC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AAH	938	MSR_MOB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ABH	939	MSR_MOB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ACH	940	MSR_PMH_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ADH	941	MSR_PMH_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AEH	942	MSR_SAA_T_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AFH	943	MSR_SAA_T_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B0H	944	MSR_U2L_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B1H	945	MSR_U2L_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B2H	946	MSR_BPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B3H	947	MSR_BPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B4H	948	MSR_IS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B5H	949	MSR_IS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B6H	950	MSR_ITLB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B7H	951	MSR_ITLB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B8H	952	MSR_CRU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B9H	953	MSR_CRU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BAH	954	MSR_IQ_ESCR0	0, 1, 2	Shared	See Section 18.6.3.1, "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 18.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.

Table 2-48. 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				
3BCH	956	MSR_RAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BDH	957	MSR_RAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BEH	958	MSR_SSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C0H	960	MSR_MS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C1H	961	MSR_MS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C2H	962	MSR_TBPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C3H	963	MSR_TBPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C4H	964	MSR_TC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C5H	965	MSR_TC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C8H	968	MSR_IX_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C9H	969	MSR_IX_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CAH	970	MSR_ALF_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CBH	971	MSR_ALF_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CCH	972	MSR_CRU_ESCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CDH	973	MSR_CRU_ESCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E0H	992	MSR_CRU_ESCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E1H	993	MSR_CRU_ESCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3F0H	1008	MSR_TC_PRECISE_EVENT	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "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 19-38.
		23:13			Reserved
		24			UOP Tag Enables replay tagging when set.

Table 2-48. 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				
		25			ENABLE_PEBS_MY_THR (R/W) Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "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 18.6.4.3, "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.
		63:27			Reserved
3F2H	1010	MSR_PEBS_MATRIX_VERT	0, 1, 2, 3, 4, 6	Shared	See Table 19-38.
400H	1024	IA32_MCO_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
401H	1025	IA32_MCO_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCI_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCI_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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_MCO_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCI_MISC MSRs." The IA32_MCO_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MCO_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."

Table 2-48. 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				
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 ADDR_V 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 MISC_V 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."
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 ADDR_V 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 MISC_V 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 ADDR_V 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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique ¹	Bit Description
Hex	Dec				
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 ADDRIV 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.
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_CTL	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_CTL	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_CTL	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.

Table 2-48. 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				
484H	1156	IA32_VMX_ENTRY_CTL5	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_CRO_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_CRO_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.
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_CTL52	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 18.6.3.4, "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.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique ¹	Bit Description
Hex	Dec				
					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 17.12, "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.
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.

Table 2-48. 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				
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 17.12, "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.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	3, 4, 6	Unique	Last Branch Record 8 See description of MSR_LASTBRANCH_0 at 6C0H.

Table 2-48. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique ¹	Bit Description
Hex	Dec				
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	Last Branch Record 9 See description of MSR_LASTBRANCH_0 at 6COH.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	Last Branch Record 10 See description of MSR_LASTBRANCH_0 at 6COH.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	Last Branch Record 11 See description of MSR_LASTBRANCH_0 at 6COH.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	Last Branch Record 12 See description of MSR_LASTBRANCH_0 at 6COH.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	Last Branch Record 13 See description of MSR_LASTBRANCH_0 at 6COH.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	Last Branch Record 14 See description of MSR_LASTBRANCH_0 at 6COH.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	Last Branch Record 15 See description of MSR_LASTBRANCH_0 at 6COH.
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.

2.19.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-49 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 2-49. 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 18.6.6, "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 18.6.6, "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)
107DOH	MSR_EFSB_DRDY0	3, 4	Shared	EFSB DRDY Event Control and Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
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 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D3H	MSR_IFSB_CNTR7	3, 4	Shared	IFSB Latency Event Counter Register (R/W) See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

The MSRs listed in Table 2-50 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 2-50 are shared between logical processors in the same core, but are replicated for each core.

Table 2-50. 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 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

Table 2-50. MSRs Unique to Intel® Xeon® Processor 7100 Series (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique	Bit Description
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 18.6.6, "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 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
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)

2.20 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 2-51. 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 2-51. 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 2.23, "MSRs in Pentium Processors," and see Table 2-2.
1H	1	P5_MC_TYPE	Unique	See Section 2.23, "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 17.17, "Time-Stamp Counter," and see Table 2-2.
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.

Table 2-51. 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			
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)
		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)			

Table 2-51. 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			
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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.15, "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:
		2:0		<ul style="list-style-type: none"> ▪ 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

Table 2-51. 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			
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	Control Register 3 Used to configure the L2 Cache.
		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	Global Machine Check 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.

Table 2-51. 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			
		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_PERFEVTSELO	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 14.8.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	Thermal Monitor Status (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	Thermal Monitor 2 Control
		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 2-51. 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			
		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 2-51. 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			
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.
200H	512	MTRRphysBase0	Unique	Memory Type Range Registers
201H	513	MTRRphysMask0	Unique	Memory Type Range Registers
202H	514	MTRRphysBase1	Unique	Memory Type Range Registers
203H	515	MTRRphysMask1	Unique	Memory Type Range Registers
204H	516	MTRRphysBase2	Unique	Memory Type Range Registers
205H	517	MTRRphysMask2	Unique	Memory Type Range Registers
206H	518	MTRRphysBase3	Unique	Memory Type Range Registers
207H	519	MTRRphysMask3	Unique	Memory Type Range Registers
208H	520	MTRRphysBase4	Unique	Memory Type Range Registers
209H	521	MTRRphysMask4	Unique	Memory Type Range Registers
20AH	522	MTRRphysBase5	Unique	Memory Type Range Registers
20BH	523	MTRRphysMask5	Unique	Memory Type Range Registers
20CH	524	MTRRphysBase6	Unique	Memory Type Range Registers
20DH	525	MTRRphysMask6	Unique	Memory Type Range Registers
20EH	526	MTRRphysBase7	Unique	Memory Type Range Registers
20FH	527	MTRRphysMask7	Unique	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Unique	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Unique	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Unique	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Unique	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Unique	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Unique	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Unique	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Unique	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Unique	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Unique	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Unique	Memory Type Range Registers

Table 2-51. 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			
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_MCO_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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 ADDR_V 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 ADDR_V 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 ADDR_V 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 ADDR_V 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 2-51. 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			
413H	1043	MSR_MC3_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
414H	1044	MSR_MC5_CTL	Unique	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
415H	1045	MSR_MC5_STATUS	Unique	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	MSR_MC5_ADDR	Unique	Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDR_V flag in the IA32_MCi_STATUS register is set.
417H	1047	MSR_MC5_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
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 5])
481H	1153	IA32_VMX_PINBASED_CTL	Unique	Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls". (If CPUID.01H:ECX.[bit 5])
482H	1154	IA32_VMX_PROCBASED_CTL	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 5])
483H	1155	IA32_VMX_EXIT_CTL	Unique	Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, "VM-Exit Controls". (If CPUID.01H:ECX.[bit 5])
484H	1156	IA32_VMX_ENTRY_CTL	Unique	Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, "VM-Entry Controls". (If CPUID.01H:ECX.[bit 5])
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 5])
486H	1158	IA32_VMX_CRO_FIXED0	Unique	Capability Reporting Register of CRO Bits Fixed to 0 (R/O) See Appendix A.7, "VMX-Fixed Bits in CRO". (If CPUID.01H:ECX.[bit 5])

Table 2-51. 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			
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 5])
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 5])
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 5])
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 5])
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	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 5] and IA32_VMX_PROCBASED_CTLSS[bit 63])
600H	1536	IA32_DS_AREA	Unique	DS Save Area (R/W) See Table 2-2. See Section 18.6.3.4, “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

2.21 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.22 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

Table 2-52. MSRs in Pentium M Processors

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.23, “MSRs in Pentium Processors.”

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
1H	1	P5_MC_TYPE	See Section 2.23, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.17, "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.
		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

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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)
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: <ul style="list-style-type: none"> ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 17.15, "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	Control Register Used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.
		63:0	Reserved
11EH	281	MSR_BBL_CR_CTL3	Control register 3 Used to configure the L2 Cache.

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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.
		22:9	Reserved
		23	L2 Not Present (RO) 0 = L2 Present 1 = L2 Not Present
		63:24	Reserved
179H	377	IA32_MCG_CAP	Read-only register that provides information about the machine-check architecture of the processor.
		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	Global Machine Check 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.

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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 14.8.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Thermal Monitor Status (R/W) See Table 2-2. See Section 14.8.2, "Thermal Monitor."
19DH	413	MSR_THERM2_CTL	Thermal Monitor 2 Control
		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

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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.
		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

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
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: <ul style="list-style-type: none"> MSR_LASTBRANCH_0_FROM_IP (at 40H). Section 17.15, "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 17.15, "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 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "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 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "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_MCO_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	See Section 14.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_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 ADDR_V 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."

Table 2-52. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name / Bit Fields	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 ADDR_V 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 ADDR_V 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 ADDR_V 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 18.6.3.4, "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

2.22 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 2-53. MSRs in the P6 Family Processors

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.23, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.23, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
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
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
		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

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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
		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)

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		27	Clock Frequency Ratio
		63:28	Reserved ¹
33H	51	MSR_TEST_CTRL	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)	Performance Counter Register See Table 2-2.
C2H	194	PerfCtr1 (PERFCTR1)	Performance Counter Register See Table 2-2.
FEH	254	MTRRcap	Memory Type Range Registers
116H	278	BBL_CR_ADDR [63:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses.
		BBL_CR_ADDR [63:32]	Reserved,
		BBL_CR_ADDR [31:3]	Address bits [35:3]
		BBL_CR_ADDR [2:0]	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	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response
		BL_CR_CTL[63:22]	Reserved
		BBL_CR_CTL[21]	Processor number ² Disable = 1 Enable = 0 Reserved

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		BBL_CR_CTL[20:19] BBL_CR_CTL[18] BBL_CR_CTL[17] BBL_CR_CTL[16] BBL_CR_CTL[15:14] BBL_CR_CTL[13:12] BBL_CR_CTL[11:10] BBL_CR_CTL[9:8] BBL_CR_CTL[7] BBL_CR_CTL[6:5] BBL_CR_CTL[4:0]	User supplied ECC Reserved L2 Hit Reserved State from L2 Modified - 11, Exclusive - 10, Shared - 01, Invalid - 00 Way from L2 Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11 Way to L2 Reserved State to L2 L2 Command 01100 Data Read w/ LRU update (RLU) 01110 Tag Read w/ Data Read (TRR) 01111 Tag Inquire (TI) 00010 L2 Control Register Read (CR) 00011 L2 Control Register Write (CW) 010 + MESI encode Tag Write w/ Data Read (TWR) 111 + MESI encode Tag Write w/ Data Write (TWW) 100 + MESI encode 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 BBL_CR_CTL3[63:26] BBL_CR_CTL3[25] BBL_CR_CTL3[24] BBL_CR_CTL3[23] BBL_CR_CTL3[22:20] 111 110 101 100 011 010 001 000 BBL_CR_CTL3[19] BBL_CR_CTL3[18]	Control register 3: used to configure the L2 Cache Reserved Cache bus fraction (read only) Reserved L2 Hardware Disable (read only) L2 Physical Address Range support 64GBytes 32GBytes 16GBytes 8GBytes 4GBytes 2GBytes 1GBytes 512MBytes Reserved Cache State error checking enable (read/write)

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13] 00001 00010 00100 01000 10000 BBL_CR_CTL3[12:11] BBL_CR_CTL3[10:9] 00 01 10 11 BBL_CR_CTL3[8] BBL_CR_CTL3[7] BBL_CR_CTL3[6] BBL_CR_CTL3[5] BBL_CR_CTL3[4:1] BBL_CR_CTL3[0]	Cache size per bank (read/write) 256KBytes 512KBytes 1MByte 2MByte 4MBytes Number of L2 banks (read only) L2 Associativity (read only) Direct Mapped 2 Way 4 Way Reserved L2 Enabled (read/write) CRTN Parity Check Enable (read/write) Address Parity Check Enable (read/write) ECC Check Enable (read/write) L2 Cache Latency (read/write) 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	Machine Check Global Control Register
17AH	378	MCG_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
17BH	379	MCG_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
186H	390	PerfEvtSel0 (EVNTSEL0)	Performance Event Select Register 0 (R/W)
		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.

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		18	E Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	PC Enabled the signaling of performance counter overflow via BPO pin
		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)	Performance Event Select for Counter 1 (R/W)
		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 BPO pin.

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
		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
		31:24	CMASK (Counter Mask)
1D9H	473	DEBUGCTLMR	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.
		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	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DCH	476	LASTBRANCHTOIP	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DDH	477	LASTINTFROMIP	Last INT from IP
1DEH	478	LASTINTTOIP	Last INT to IP
200H	512	MTRRphysBase0	Memory Type Range Registers
201H	513	MTRRphysMask0	Memory Type Range Registers
202H	514	MTRRphysBase1	Memory Type Range Registers
203H	515	MTRRphysMask1	Memory Type Range Registers
204H	516	MTRRphysBase2	Memory Type Range Registers
205H	517	MTRRphysMask2	Memory Type Range Registers
206H	518	MTRRphysBase3	Memory Type Range Registers
207H	519	MTRRphysMask3	Memory Type Range Registers
208H	520	MTRRphysBase4	Memory Type Range Registers
209H	521	MTRRphysMask4	Memory Type Range Registers
20AH	522	MTRRphysBase5	Memory Type Range Registers

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
20BH	523	MTRRphysMask5	Memory Type Range Registers
20CH	524	MTRRphysBase6	Memory Type Range Registers
20DH	525	MTRRphysMask6	Memory Type Range Registers
20EH	526	MTRRphysBase7	Memory Type Range Registers
20FH	527	MTRRphysMask7	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Memory Type Range Registers
2FFH	767	MTRRdefType	Memory Type Range Registers
		2:0	Default memory type
		10	Fixed MTRR enable
		11	MTRR Enable
400H	1024	MCO_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
401H	1025	MCO_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCI_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
		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 MCO_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
63	MC_STATUS_V		
402H	1026	MCO_ADDR	
403H	1027	MCO_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

Table 2-53. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name / Bit Fields	Bit Description
Hex	Dec		
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MCO_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 MCO_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 MCO_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.
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 MCO_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.

2.23 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 2-54. 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 17.17, "Time-Stamp Counter."
11H	17	CESR	See Section 18.6.9.1, "Control and Event Select Register (CESR)."
12H	18	CTRO	Section 18.6.9.3, "Events Counted."
13H	19	CTR1	Section 18.6.9.3, "Events Counted."

2.24 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 2-48
MSR_ALF_ESCR1	
0FH	See Table 2-48
MSR_ANY_CORE_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_ANY_GFXE_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_BO_PMON_BOX_CTRL	
06_2EH	See Table 2-17
MSR_BO_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_BO_PMON_BOX_STATUS	
06_2EH	See Table 2-17
MSR_BO_PMON_CTRO	
06_2EH	See Table 2-17
MSR_BO_PMON_CTR1	
06_2EH	See Table 2-17
MSR_BO_PMON_CTR2	
06_2EH	See Table 2-17
MSR_BO_PMON_CTR3	
06_2EH	See Table 2-17
MSR_BO_PMON_EVNT_SELO	
06_2EH	See Table 2-17
MSR_BO_PMON_EVNT_SEL1	
06_2EH	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_BO_PMON_EVNT_SEL2 06_2EH	See Table 2-17
MSR_BO_PMON_EVNT_SEL3 06_2EH	See Table 2-17
MSR_BO_PMON_MASK 06_2EH	See Table 2-17
MSR_BO_PMON_MATCH 06_2EH	See Table 2-17
MSR_B1_PMON_BOX_CTRL 06_2EH	See Table 2-17
MSR_B1_PMON_BOX_OVF_CTRL 06_2EH	See Table 2-17
MSR_B1_PMON_BOX_STATUS 06_2EH	See Table 2-17
MSR_B1_PMON_CTRL0 06_2EH	See Table 2-17
MSR_B1_PMON_CTRL1 06_2EH	See Table 2-17
MSR_B1_PMON_CTRL2 06_2EH	See Table 2-17
MSR_B1_PMON_CTRL3 06_2EH	See Table 2-17
MSR_B1_PMON_EVNT_SELO 06_2EH	See Table 2-17
MSR_B1_PMON_EVNT_SEL1 06_2EH	See Table 2-17
MSR_B1_PMON_EVNT_SEL2 06_2EH	See Table 2-17
MSR_B1_PMON_EVNT_SEL3 06_2EH	See Table 2-17
MSR_B1_PMON_MASK 06_2EH	See Table 2-17
MSR_B1_PMON_MATCH 06_2EH	See Table 2-17
MSR_BBL_CR_CTL 06_09H	See Table 2-52
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 2-51
06_09H	See Table 2-52

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_BIOS_DONE	
06_7DH, 06_7EH	See Table 2-44
MSR_BPU_CCCR0	
OFH	See Table 2-48
MSR_BPU_CCCR1	
OFH	See Table 2-48
MSR_BPU_CCCR2	
OFH	See Table 2-48
MSR_BPU_CCCR3	
OFH	See Table 2-48
MSR_BPU_COUNTER0	
OFH	See Table 2-48
MSR_BPU_COUNTER1	
OFH	See Table 2-48
MSR_BPU_COUNTER2	
OFH	See Table 2-48
MSR_BPU_COUNTER3	
OFH	See Table 2-48
MSR_BPU_ESCR0	
OFH	See Table 2-48
MSR_BPU_ESCR1	
OFH	See Table 2-48
MSR_BR_DETECT_COUNTER_CONFIG_i	
06_66H	See Table 2-42
MSR_BR_DETECT_CTRL	
06_66H	See Table 2-42
MSR_BR_DETECT_STATUS	
06_66H	See Table 2-42
MSR_BSU_ESCR0	
OFH	See Table 2-48
MSR_BSU_ESCR1	
OFH	See Table 2-48
MSR_CO_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_CO_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_CO_PMON_BOX_FILTER1	
06_3EH	See Table 2-28

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_CO_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_CO_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL4	
06_2EH	See Table 2-17
MSR_CO_PMON_CTRL5	
06_2EH	See Table 2-17
MSR_CO_PMON_EVNT_SELO	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_CTRL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_CO_PMON_CTR2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_CO_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_CO_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C1_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C1_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C1_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C1_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C1_PMON_CTR0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_CTR1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_CTR2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_CTR3	
06_2EH	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_CTR4	
06_2EH	See Table 2-17
MSR_C1_PMON_CTR5	
06_2EH	See Table 2-17
MSR_C1_PMON_EVNT_SELO	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C1_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C1_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C10_PMON_BOX_FILTER	
06_3EH	See Table 2-28
MSR_C10_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C10_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C11_PMON_BOX_FILTER	
06_3EH	See Table 2-28
MSR_C11_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C11_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C12_PMON_BOX_FILTER	
06_3EH	See Table 2-28

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C12_PMON_BOX_FILTER0 06_3FH	See Table 2-33
MSR_C12_PMON_BOX_FILTER1 06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C13_PMON_BOX_FILTER 06_3EH	See Table 2-28
MSR_C13_PMON_BOX_FILTER0 06_3FH	See Table 2-33
MSR_C13_PMON_BOX_FILTER1 06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C14_PMON_BOX_FILTER 06_3EH	See Table 2-28
MSR_C14_PMON_BOX_FILTER0 06_3FH	See Table 2-33
MSR_C14_PMON_BOX_FILTER1 06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C15_PMON_BOX_CTL 06_3FH	See Table 2-33
MSR_C15_PMON_BOX_FILTER0 06_3FH	See Table 2-33
MSR_C15_PMON_BOX_FILTER1 06_3FH	See Table 2-33
MSR_C15_PMON_BOX_STATUS 06_3FH	See Table 2-33
MSR_C15_PMON_CTR0 06_3FH	See Table 2-33
MSR_C15_PMON_CTR1 06_3FH	See Table 2-33
MSR_C15_PMON_CTR2 06_3FH	See Table 2-33
MSR_C15_PMON_CTR3 06_3FH	See Table 2-33
MSR_C15_PMON_EVNTSELO 06_3FH	See Table 2-33
MSR_C15_PMON_EVNTSEL1 06_3FH	See Table 2-33
MSR_C15_PMON_EVNTSEL2 06_3FH	See Table 2-33
MSR_C15_PMON_EVNTSEL3	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_C16_PMON_BOX_CTL	
06_3FH	See Table 2-33
MSR_C16_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C16_PMON_BOX_FILTER1	
06_3FH	See Table 2-33
MSR_C16_PMON_BOX_STATUS	
06_3FH	See Table 2-33
MSR_C16_PMON_CTRL0	
06_3FH	See Table 2-33
MSR_C16_PMON_CTRL3	
06_3FH	See Table 2-33
MSR_C16_PMON_CTRL2	
06_3FH	See Table 2-33
MSR_C16_PMON_CTRL3	
06_3FH	See Table 2-33
MSR_C16_PMON_EVNTSELO	
06_3FH	See Table 2-33
MSR_C16_PMON_EVNTSEL1	
06_3FH	See Table 2-33
MSR_C16_PMON_EVNTSEL2	
06_3FH	See Table 2-33
MSR_C16_PMON_EVNTSEL3	
06_3FH	See Table 2-33
MSR_C17_PMON_BOX_CTL	
06_3FH	See Table 2-33
MSR_C17_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C17_PMON_BOX_FILTER1	
06_3FH	See Table 2-33
MSR_C17_PMON_BOX_STATUS	
06_3FH	See Table 2-33
MSR_C17_PMON_CTRL0	
06_3FH	See Table 2-33
MSR_C17_PMON_CTRL1	
06_3FH	See Table 2-33
MSR_C17_PMON_CTRL2	
06_3FH	See Table 2-33
MSR_C17_PMON_CTRL3	
06_3FH	See Table 2-33
MSR_C17_PMON_EVNTSELO	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_C17_PMON_EVNTSEL1	
06_3FH	See Table 2-33
MSR_C17_PMON_EVNTSEL2	
06_3FH	See Table 2-33
MSR_C17_PMON_EVNTSEL3	
06_3FH	See Table 2-33
MSR_C2_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C2_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C2_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C2_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C2_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C2_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_CTRL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_CTRL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_CTRL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_CTRL4	
06_2EH	See Table 2-17
MSR_C2_PMON_CTRL5	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
MSR_C2_PMON_EVNT_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C2_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C2_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C3_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C3_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C3_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C3_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C3_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C3_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_CTRL1	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_CTR2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_CTR3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_CTR4	
06_2EH	See Table 2-17
MSR_C3_PMON_CTR5	
06_2EH	See Table 2-17
MSR_C3_PMON_EVNT_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C3_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C3_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C4_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C4_PMON_BOX_FILTER0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_C4_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C4_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C4_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C4_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_CTRL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_CTRL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_CTRL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_CTRL4	
06_2EH	See Table 2-17
MSR_C4_PMON_CTRL5	
06_2EH	See Table 2-17
MSR_C4_PMON_EVTN_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_EVTN_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_EVTN_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_EVTN_SEL3	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C4_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C4_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C5_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C5_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C5_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C5_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C5_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C5_PMON_CTR0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_CTR1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_CTR2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_CTR3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_CTR4	
06_2EH	See Table 2-17
MSR_C5_PMON_CTR5	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
MSR_C5_PMON_EVNT_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C5_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C5_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C6_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C6_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C6_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C6_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C6_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C6_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_CTRL1	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_CTR2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_CTR3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_CTR4	
06_2EH	See Table 2-17
MSR_C6_PMON_CTR5	
06_2EH	See Table 2-17
MSR_C6_PMON_EVNT_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C6_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C6_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C7_PMON_BOX_CTRL	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_BOX_FILTER	
06_2DH	See Table 2-24
MSR_C7_PMON_BOX_FILTER0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_C7_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C7_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_C7_PMON_BOX_STATUS	
06_2EH	See Table 2-17
06_3FH	See Table 2-33
MSR_C7_PMON_CTRL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_CTRL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_CTRL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_CTRL3	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_CTRL4	
06_2EH	See Table 2-17
MSR_C7_PMON_CTRL5	
06_2EH	See Table 2-17
MSR_C7_PMON_EVTN_SEL0	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_EVTN_SEL1	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_EVTN_SEL2	
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_EVTN_SEL3	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-17
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_C7_PMON_EVNT_SEL4	
06_2EH	See Table 2-17
MSR_C7_PMON_EVNT_SEL5	
06_2EH	See Table 2-17
MSR_C8_PMON_BOX_CTRL	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_BOX_FILTER	
06_3EH	See Table 2-28
MSR_C8_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C8_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-19
MSR_C8_PMON_BOX_STATUS	
06_2FH	See Table 2-19
06_3FH	See Table 2-33
MSR_C8_PMON_CTR0	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_CTR1	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_CTR2	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_CTR3	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_CTR4	
06_2FH	See Table 2-19
MSR_C8_PMON_CTR5	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2FH	See Table 2-19
MSR_C8_PMON_EVNT_SEL0	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_EVNT_SEL1	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_EVNT_SEL2	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_EVNT_SEL3	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C8_PMON_EVNT_SEL4	
06_2FH	See Table 2-19
MSR_C8_PMON_EVNT_SEL5	
06_2FH	See Table 2-19
MSR_C9_PMON_BOX_CTRL	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_BOX_FILTER	
06_3EH	See Table 2-28
MSR_C9_PMON_BOX_FILTER0	
06_3FH	See Table 2-33
MSR_C9_PMON_BOX_FILTER1	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-19
MSR_C9_PMON_BOX_STATUS	
06_2FH	See Table 2-19
06_3FH	See Table 2-33
MSR_C9_PMON_CTRL0	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_CTRL1	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_CTR2	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_CTR3	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_CTR4	
06_2FH	See Table 2-19
MSR_C9_PMON_CTR5	
06_2FH	See Table 2-19
MSR_C9_PMON_EVNT_SEL0	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_EVNT_SEL1	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_EVNT_SEL2	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_EVNT_SEL3	
06_2FH	See Table 2-19
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_C9_PMON_EVNT_SEL4	
06_2FH	See Table 2-19
MSR_C9_PMON_EVNT_SEL5	
06_2FH	See Table 2-19
MSR_CC6_DEMOTION_POLICY_CONFIG	
06_37H	See Table 2-9
MSR_CONFIG_TDP_CONTROL	
06_3AH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H, 06_85H	See Table 2-46
MSR_CONFIG_TDP_LEVEL1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3AH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H, 06_85H.....	See Table 2-46
MSR_CONFIG_TDP_LEVEL2	
06_3AH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H, 06_85H	See Table 2-46
MSR_CONFIG_TDP_NOMINAL	
06_3AH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H, 06_85H	See Table 2-46
MSR_CORE_C1_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_66H	See Table 2-42
MSR_CORE_C3_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH.....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
MSR_CORE_C6_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH.....	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
MSR_CORE_C7_RESIDENCY	
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 2-20
MSR_CORE_GFXE_OVERLAP_CO	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_CORE_HDC_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_CORE_PERF_LIMIT_REASONS	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-30
06_3F	See Table 2-32
06_56H, 06_4FH	See Table 2-36
06_57H, 06_85H.....	See Table 2-46
MSR_CORE_THREAD_COUNT	
06_3FH.....	See Table 2-32
MSR_CRASHLOG_CONTROL	
06_7DH, 06_7EH.....	See Table 2-44
MSR_CRU_ESCR0	
0FH	See Table 2-48
MSR_CRU_ESCR1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-48
MSR_CRU_ESCR2	
0FH	See Table 2-48
MSR_CRU_ESCR3	
0FH	See Table 2-48
MSR_CRU_ESCR4	
0FH	See Table 2-48
MSR_CRU_ESCR5	
0FH	See Table 2-48
MSR_DAC_ESCR0	
0FH	See Table 2-48
MSR_DAC_ESCR1	
0FH	See Table 2-48
MSR_DRAM_ENERGY_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-23
06_3EH, 06_3FH	See Table 2-26
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3F	See Table 2-32
06_56H, 06_4FH	See Table 2-36
06_57H, 06_85H	See Table 2-46
MSR_DRAM_PERF_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-23
06_3EH, 06_3FH	See Table 2-26
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3F	See Table 2-32
06_56H, 06_4FH	See Table 2-36
06_57H, 06_85H	See Table 2-46
MSR_DRAM_POWER_INFO	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-23
06_3EH, 06_3FH	See Table 2-26
06_3F	See Table 2-32
06_56H, 06_4FH	See Table 2-36
06_57H, 06_85H	See Table 2-46
MSR_DRAM_POWER_LIMIT	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-23
06_3EH, 06_3FH	See Table 2-26
06_3F	See Table 2-32
06_56H, 06_4FH	See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_57H, 06_85H	See Table 2-46
MSR_EBC_FREQUENCY_ID	
0FH	See Table 2-48
MSR_EBC_HARD_POWERON	
0FH	See Table 2-48
MSR_EBC_SOFT_POWERON	
0FH	See Table 2-48
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, 06_5CH, 06_7AH	See Table 2-6
06_0EH	See Table 2-51
06_09H	See Table 2-52
MSR_EFSB_DRDY0	
0F_03H, 0F_04H	See Table 2-49
MSR_EFSB_DRDY1	
0F_03H, 0F_04H	See Table 2-49
MSR_EMON_L3_CTR_CTL0	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL1	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL2	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL3	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL4	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL5	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL6	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_CTR_CTL7	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-50
MSR_EMON_L3_GL_CTL	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0FH, 06_17H	See Table 2-3
MSR_ERROR_CONTROL	
06_2DH	See Table 2-23
06_3EH	See Table 2-26
06_3F	See Table 2-32
MSR_FAST_UNCORE_MSRS_CAPABILITY	
06_7DH, 06_7EH	See Table 2-44
MSR_FAST_UNCORE_MSRS_CTL	
06_7DH, 06_7EH	See Table 2-44
MSR_FAST_UNCORE_MSRS_STATUS	
06_7DH, 06_7EH	See Table 2-44
MSR_FEATURE_CONFIG	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_25H, 06_2CH	See Table 2-18
06_2FH	See Table 2-19
06_2AH, 06_2DH	See Table 2-20
06_57H, 06_85H	See Table 2-46
MSR_FIRM_ESCRO	
0FH	See Table 2-48
MSR_FIRM_ESCR1	
0FH	See Table 2-48
MSR_FLAME_CCCRO	
0FH	See Table 2-48
MSR_FLAME_CCCR1	
0FH	See Table 2-48
MSR_FLAME_CCCR2	
0FH	See Table 2-48
MSR_FLAME_CCCR3	
0FH	See Table 2-48
MSR_FLAME_COUNTER0	
0FH	See Table 2-48
MSR_FLAME_COUNTER1	
0FH	See Table 2-48
MSR_FLAME_COUNTER2	
0FH	See Table 2-48
MSR_FLAME_COUNTER3	
0FH	See Table 2-48
MSR_FLAME_ESCRO	
0FH	See Table 2-48
MSR_FLAME_ESCR1	
0FH	See Table 2-48
MSR_FSB_ESCRO	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-48
MSR_FSB_ESCR1	
0FH	See Table 2-48
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 2-51
MSR_GQ_SNOOP_MESF	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_GRAPHICS_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H	See Table 2-30
MSR_IFSB_BUSQ0	
0F_03H, 0F_04H	See Table 2-49
MSR_IFSB_BUSQ1	
0F_03H, 0F_04H	See Table 2-49
MSR_IFSB_CNTR7	
0F_03H, 0F_04H	See Table 2-49
MSR_IFSB_CTL6	
0F_03H, 0F_04H	See Table 2-49
MSR_IFSB_SNPQ0	
0F_03H, 0F_04H	See Table 2-49
MSR_IFSB_SNPQ1	
0F_03H, 0F_04H	See Table 2-49
MSR_IQ_CCCR0	
0FH	See Table 2-48
MSR_IQ_CCCR1	
0FH	See Table 2-48
MSR_IQ_CCCR2	
0FH	See Table 2-48
MSR_IQ_CCCR3	
0FH	See Table 2-48
MSR_IQ_CCCR4	
0FH	See Table 2-48
MSR_IQ_CCCR5	
0FH	See Table 2-48
MSR_IQ_COUNTER0	
0FH	See Table 2-48
MSR_IQ_COUNTER1	
0FH	See Table 2-48
MSR_IQ_COUNTER2	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-48
MSR_IQ_COUNTER3	
0FH	See Table 2-48
MSR_IQ_COUNTER4	
0FH	See Table 2-48
MSR_IQ_COUNTER5	
0FH	See Table 2-48
MSR_IQ_ESCR0	
0FH	See Table 2-48
MSR_IQ_ESCR1	
0FH	See Table 2-48
MSR_IS_ESCR0	
0FH	See Table 2-48
MSR_IS_ESCR1	
0FH	See Table 2-48
MSR_ITLB_ESCR0	
0FH	See Table 2-48
MSR_ITLB_ESCR1	
0FH	See Table 2-48
MSR_IX_ESCR0	
0FH	See Table 2-48
MSR_IX_ESCR1	
0FH	See Table 2-48
MSR_LASTBRANCH_0	
0FH	See Table 2-48
06_0EH	See Table 2-51
06_09H	See Table 2-52
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
06_7AH	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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_7AH	See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_10_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_10_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_11_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_11_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_12_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_12_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_13_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_13_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_14_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_14_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_15_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_15_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_16_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_16_TO_IP	
06_5CH, 06_7AH	See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_17_FROM_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_17_TO_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_17_TO_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_18_FROM_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_18_FROM_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_18_TO_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_18_TO_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_19_FROM_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_19_FROM_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_19_TO_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_19_TO_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_2 0FH	See Table 2-48
MSR_LASTBRANCH_2 06_0EH	See Table 2-51
MSR_LASTBRANCH_2 06_09H	See Table 2-52
MSR_LASTBRANCH_2_FROM_IP 06_0FH, 06_17H	See Table 2-3
MSR_LASTBRANCH_2_FROM_IP 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
MSR_LASTBRANCH_2_FROM_IP 06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
MSR_LASTBRANCH_2_FROM_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_2_FROM_IP 06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
MSR_LASTBRANCH_2_FROM_IP 06_2AH, 06_2DH	See Table 2-20
MSR_LASTBRANCH_2_FROM_IP 0FH	See Table 2-48
MSR_LASTBRANCH_2_TO_IP 06_0FH, 06_17H	See Table 2-3
MSR_LASTBRANCH_2_TO_IP 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
MSR_LASTBRANCH_2_TO_IP 06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
MSR_LASTBRANCH_2_TO_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_2_TO_IP 06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
MSR_LASTBRANCH_2_TO_IP 06_2AH, 06_2DH	See Table 2-20
MSR_LASTBRANCH_2_TO_IP 0FH	See Table 2-48
MSR_LASTBRANCH_20_FROM_IP 06_5CH, 06_7AH	See Table 2-12
MSR_LASTBRANCH_20_FROM_IP 06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_20_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_21_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_21_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_22_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_22_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_23_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_23_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_24_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_24_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_25_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_25_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_26_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_26_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_27_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_27_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_28_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_28_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_29_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_29_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_3	
0FH	See Table 2-48
06_0EH	See Table 2-51
06_09H	See Table 2-52
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_30_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_30_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_31_FROM_IP	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_31_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_LASTBRANCH_4	
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH.....	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH.....	See Table 2-48
MSR_LASTBRANCH_5	
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH.....	See Table 2-48
MSR_LASTBRANCH_5_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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH.....	See Table 2-48
MSR_LASTBRANCH_6	
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52

MSR Name and CPUID DisplayFamily_DisplayModel	Location
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_7	
06_0EH	See Table 2-51
06_09H	See Table 2-52
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_8_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_8_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_9_FROM_IP	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
MSR_LASTBRANCH_9_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
0FH	See Table 2-48
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_57H, 06_85H	See Table 2-46
06_0EH	See Table 2-51
06_09H	See Table 2-52
MSR_LASTBRANCH_INFO_0	
06_7AH	See Table 2-13
MSR_LBR_INFO_1	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_10	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_11	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_12	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_13	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_14	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
06_7AH	See Table 2-13
MSR_LBR_INFO_15	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_7AH.....	See Table 2-13
MSR_LBR_INFO_16	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_17	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_18	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_19	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_2	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_20	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_21	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_22	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_23	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_24	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_25	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_26	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_27	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_28	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_7AH.....	See Table 2-13
MSR_LBR_INFO_29	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_3	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_30	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_31	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_4	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_5	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_6	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_7	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_8	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_INFO_9	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
06_7AH.....	See Table 2-13
MSR_LBR_SELECT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 2-7
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3CH, 06_45H, 06_46H.....	See Table 2-29
06_57H, 06_85H.....	See Table 2-46
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, 06_5CH, 06_7AH.....	See Table 2-6

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
0FH	See Table 2-48
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
0FH	See Table 2-48
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
MSR_MO_PMON_ADDR_MASK	
06_2EH.....	See Table 2-17
MSR_MO_PMON_ADDR_MATCH	
06_2EH.....	See Table 2-17
MSR_MO_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
MSR_MO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_MO_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR0	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR1	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR2	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR3	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR4	
06_2EH.....	See Table 2-17
MSR_MO_PMON_CTR5	
06_2EH.....	See Table 2-17
MSR_MO_PMON_DSP	
06_2EH.....	See Table 2-17
MSR_MO_PMON_EVNT_SELO	
06_2EH.....	See Table 2-17

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M0_PMON_EVT_SEL1 06_2EH.....	See Table 2-17
MSR_M0_PMON_EVT_SEL2 06_2EH.....	See Table 2-17
MSR_M0_PMON_EVT_SEL3 06_2EH.....	See Table 2-17
MSR_M0_PMON_EVT_SEL4 06_2EH.....	See Table 2-17
MSR_M0_PMON_EVT_SEL5 06_2EH.....	See Table 2-17
MSR_M0_PMON_ISS 06_2EH.....	See Table 2-17
MSR_M0_PMON_MAP 06_2EH.....	See Table 2-17
MSR_M0_PMON_MM_CONFIG 06_2EH.....	See Table 2-17
MSR_M0_PMON_MSC_THR 06_2EH.....	See Table 2-17
MSR_M0_PMON_PGT 06_2EH.....	See Table 2-17
MSR_M0_PMON_PLD 06_2EH.....	See Table 2-17
MSR_M0_PMON_TIMESTAMP 06_2EH.....	See Table 2-17
MSR_M0_PMON_ZDP 06_2EH.....	See Table 2-17
MSR_M1_PMON_ADDR_MASK 06_2EH.....	See Table 2-17
MSR_M1_PMON_ADDR_MATCH 06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_CTRL 06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_OVF_CTRL 06_2EH.....	See Table 2-17
MSR_M1_PMON_BOX_STATUS 06_2EH.....	See Table 2-17
MSR_M1_PMON_CTR0 06_2EH.....	See Table 2-17
MSR_M1_PMON_CTR1 06_2EH.....	See Table 2-17
MSR_M1_PMON_CTR2 06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M1_PMON_CTR3 06_2EH.....	See Table 2-17
MSR_M1_PMON_CTR4 06_2EH.....	See Table 2-17
MSR_M1_PMON_CTR5 06_2EH.....	See Table 2-17
MSR_M1_PMON_DSP 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SELO 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL1 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL2 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL3 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL4 06_2EH.....	See Table 2-17
MSR_M1_PMON_EVNT_SEL5 06_2EH.....	See Table 2-17
MSR_M1_PMON_ISS 06_2EH.....	See Table 2-17
MSR_M1_PMON_MAP 06_2EH.....	See Table 2-17
MSR_M1_PMON_MM_CONFIG 06_2EH.....	See Table 2-17
MSR_M1_PMON_MSC_THR 06_2EH.....	See Table 2-17
MSR_M1_PMON_PGT 06_2EH.....	See Table 2-17
MSR_M1_PMON_PLD 06_2EH.....	See Table 2-17
MSR_M1_PMON_TIMESTAMP 06_2EH.....	See Table 2-17
MSR_M1_PMON_ZDP 06_2EH.....	See Table 2-17
IA32_MCO_MISC / MSR_MCO_MISC 06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
MSR_MCO_RESIDENCY 06_57H, 06_85H.....	See Table 2-46
IA32_MC1_MISC / MSR_MC1_MISC 06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC10_ADDR / MSR_MC10_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC10_CTL / MSR_MC10_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC10_MISC / MSR_MC10_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC10_STATUS / MSR_MC10_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC11_ADDR / MSR_MC11_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC11_CTL / MSR_MC11_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC11_MISC / MSR_MC11_MISC	
06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC11_STATUS / MSR_MC11_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC12_ADDR / MSR_MC12_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC12_CTL / MSR_MC12_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC12_MISC / MSR_MC12_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC12_STATUS / MSR_MC12_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC13_ADDR / MSR_MC13_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC13_CTL / MSR_MC13_CTL	
06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC13_MISC / MSR_MC13_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC13_STATUS / MSR_MC13_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC14_ADDR / MSR_MC14_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC14_CTL / MSR_MC14_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC14_MISC / MSR_MC14_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC14_STATUS / MSR_MC14_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC15_ADDR / MSR_MC15_ADDR	
06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC15_CTL / MSR_MC15_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC15_MISC / MSR_MC15_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC15_STATUS / MSR_MC15_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC16_ADDR / MSR_MC16_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC16_CTL / MSR_MC16_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC16_MISC / MSR_MC16_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC16_STATUS / MSR_MC16_STATUS	
06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC17_ADDR / MSR_MC17_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC17_CTL / MSR_MC17_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC17_MISC / MSR_MC17_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC17_STATUS / MSR_MC17_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC18_ADDR / MSR_MC18_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC18_CTL / MSR_MC18_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC18_MISC / MSR_MC18_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC18_STATUS / MSR_MC18_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC19_ADDR / MSR_MC19_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC19_CTL / MSR_MC19_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC19_MISC / MSR_MC19_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC19_STATUS / MSR_MC19_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC2_MISC / MSR_MC2_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
IA32_MC20_ADDR / MSR_MC20_ADDR	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC20_CTL / MSR_MC20_CTL	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC20_MISC / MSR_MC20_MISC	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC20_STATUS / MSR_MC20_STATUS	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC21_ADDR / MSR_MC21_ADDR	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_CTL / MSR_MC21_CTL	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_MISC / MSR_MC21_MISC	
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC21_STATUS / MSR_MC21_STATUS	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4F.....	See Table 2-38
IA32_MC22_ADDR / MSR_MC22_ADDR	
06_3EH.....	See Table 2-26
IA32_MC22_CTL / MSR_MC22_CTL	
06_3EH.....	See Table 2-26
IA32_MC22_MISC / MSR_MC22_MISC	
06_3EH.....	See Table 2-26
IA32_MC22_STATUS / MSR_MC22_STATUS	
06_3EH.....	See Table 2-26
IA32_MC23_ADDR / MSR_MC23_ADDR	
06_3EH.....	See Table 2-26
IA32_MC23_CTL / MSR_MC23_CTL	
06_3EH.....	See Table 2-26
IA32_MC23_MISC / MSR_MC23_MISC	
06_3EH.....	See Table 2-26
IA32_MC23_STATUS / MSR_MC23_STATUS	
06_3EH.....	See Table 2-26
IA32_MC24_ADDR / MSR_MC24_ADDR	
06_3EH.....	See Table 2-26
IA32_MC24_CTL / MSR_MC24_CTL	
06_3EH.....	See Table 2-26
IA32_MC24_MISC / MSR_MC24_MISC	
06_3EH.....	See Table 2-26
IA32_MC24_STATUS / MSR_MC24_STATUS	
06_3EH.....	See Table 2-26
IA32_MC25_ADDR / MSR_MC25_ADDR	
06_3EH.....	See Table 2-26
IA32_MC25_CTL / MSR_MC25_CTL	
06_3EH.....	See Table 2-26
IA32_MC25_MISC / MSR_MC25_MISC	
06_3EH.....	See Table 2-26
IA32_MC25_STATUS / MSR_MC25_STATUS	
06_3EH.....	See Table 2-26
IA32_MC26_ADDR / MSR_MC26_ADDR	
06_3EH.....	See Table 2-26
IA32_MC26_CTL / MSR_MC26_CTL	
06_3EH.....	See Table 2-26
IA32_MC26_MISC / MSR_MC26_MISC	
06_3EH.....	See Table 2-26

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC26_STATUS / MSR_MC26_STATUS 06_3EH.....	See Table 2-26
IA32_MC27_ADDR / MSR_MC27_ADDR 06_3EH.....	See Table 2-26
IA32_MC27_CTL / MSR_MC27_CTL 06_3EH.....	See Table 2-26
IA32_MC27_MISC / MSR_MC27_MISC 06_3EH.....	See Table 2-26
IA32_MC27_STATUS / MSR_MC27_STATUS 06_3EH.....	See Table 2-26
IA32_MC28_ADDR / MSR_MC28_ADDR 06_3EH.....	See Table 2-26
IA32_MC28_CTL / MSR_MC28_CTL 06_3EH.....	See Table 2-26
IA32_MC28_MISC / MSR_MC28_MISC 06_3EH.....	See Table 2-26
IA32_MC28_STATUS / MSR_MC28_STATUS 06_3EH.....	See Table 2-26
IA32_MC29_ADDR / MSR_MC29_ADDR 06_3EH.....	See Table 2-27
IA32_MC29_CTL / MSR_MC29_CTL 06_3EH.....	See Table 2-27
IA32_MC29_MISC / MSR_MC29_MISC 06_3EH.....	See Table 2-27
IA32_MC29_STATUS / MSR_MC29_STATUS 06_3EH.....	See Table 2-27
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC3_MISC / MSR_MC3_MISC	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_0EH	See Table 2-51
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_57H, 06_85H	See Table 2-46
06_0EH	See Table 2-51
06_09H	See Table 2-52
IA32_MC30_ADDR / MSR_MC30_ADDR	
06_3EH	See Table 2-27
IA32_MC30_CTL / MSR_MC30_CTL	
06_3EH	See Table 2-27
IA32_MC30_MISC / MSR_MC30_MISC	
06_3EH	See Table 2-27
IA32_MC30_STATUS / MSR_MC30_STATUS	
06_3EH	See Table 2-27
IA32_MC31_ADDR / MSR_MC31_ADDR	
06_3EH	See Table 2-27
IA32_MC31_CTL / MSR_MC31_CTL	
06_3EH	See Table 2-27
IA32_MC31_MISC / MSR_MC31_MISC	
06_3EH	See Table 2-27
IA32_MC31_STATUS / MSR_MC31_STATUS	
06_3EH	See Table 2-27
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_57H, 06_85H	See Table 2-46
06_0EH	See Table 2-51
06_09H	See Table 2-52
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, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_57H, 06_85H	See Table 2-46

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
IA32_MC4_CTL2 / MSR_MC4_CTL2	
06_2AH, 06_2DH.....	See Table 2-20
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, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
MSR_MC5_ADDR / MSR_MC5_ADDR	
06_0FH, 06_17H.....	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
IA32_MC5_CTL / MSR_MC5_CTL	
06_0FH, 06_17H.....	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
IA32_MC5_MISC / MSR_MC5_MISC	
06_0FH, 06_17H.....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_0EH.....	See Table 2-51
IA32_MC5_STATUS / MSR_MC5_STATUS	
06_0FH, 06_17H.....	See Table 2-3

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3FH.....	See Table 2-32
06_4FH.....	See Table 2-38
06_57H, 06_85H.....	See Table 2-46
06_0EH.....	See Table 2-51
IA32_MC6_ADDR / MSR_MC6_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC6_CTL / MSR_MC6_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
MSR_MC6_DEMOTION_POLICY_CONFIG	
06_37H.....	See Table 2-9
IA32_MC6_MISC / MSR_MC6_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
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, 06_85H.....	See Table 2-46
IA32_CORE_CAPABILITIES (Note there are no architecturally defined bits; all bits are model-specific)	
06_86H.....	See Table 2-14
IA32_MC6_STATUS / MSR_MC6_STATUS	
06_0FH, 06_17H.....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC7_ADDR / MSR_MC7_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC7_CTL / MSR_MC7_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC7_MISC / MSR_MC7_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC7_STATUS / MSR_MC7_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC8_ADDR / MSR_MC8_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC8_CTL / MSR_MC8_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH.....	See Table 2-38
IA32_MC8_MISC / MSR_MC8_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC8_STATUS / MSR_MC8_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_4FH.....	See Table 2-38
IA32_MC9_ADDR / MSR_MC9_ADDR	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC9_CTL / MSR_MC9_CTL	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC9_MISC / MSR_MC9_MISC	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
IA32_MC9_STATUS / MSR_MC9_STATUS	
06_2EH.....	See Table 2-17
06_2DH.....	See Table 2-23
06_3EH.....	See Table 2-26
06_3F.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-37
06_4FH.....	See Table 2-38
MSR_MCG_MISC	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 2-48
MSR_MCG_R10	
0FH.....	See Table 2-48
MSR_MCG_R11	
0FH.....	See Table 2-48
MSR_MCG_R12	
0FH.....	See Table 2-48
MSR_MCG_R13	
0FH.....	See Table 2-48
MSR_MCG_R14	
0FH.....	See Table 2-48
MSR_MCG_R15	
0FH.....	See Table 2-48
MSR_MCG_R8	
0FH.....	See Table 2-48
MSR_MCG_R9	
0FH.....	See Table 2-48
MSR_MCG_RAX	
0FH.....	See Table 2-48
MSR_MCG_RBP	
0FH.....	See Table 2-48
MSR_MCG_RBX	
0FH.....	See Table 2-48
MSR_MCG_RCX	
0FH.....	See Table 2-48
MSR_MCG_RDI	
0FH.....	See Table 2-48
MSR_MCG_RDX	
0FH.....	See Table 2-48
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5	
0FH.....	See Table 2-48
MSR_MCG_RFLAGS	
0FH.....	See Table 2-48
MSR_MCG_RIP	
0FH.....	See Table 2-48
MSR_MCG_RSI	
0FH.....	See Table 2-48
MSR_MCG_RSP	
0FH.....	See Table 2-48
MSR_MISC_FEATURE_CONTROL	
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH	See Table 2-20
MSR_MISC_PWR_MGMT	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
MSR_MOB_ESCRO	
0FH	See Table 2-48
MSR_MOB_ESCR1	
0FH	See Table 2-48
MSR_MS_CCCRO	
0FH	See Table 2-48
MSR_MS_CCCR1	
0FH	See Table 2-48
MSR_MS_CCCR2	
0FH	See Table 2-48
MSR_MS_CCCR3	
0FH	See Table 2-48
MSR_MS_COUNTER0	
0FH	See Table 2-48
MSR_MS_COUNTER1	
0FH	See Table 2-48
MSR_MS_COUNTER2	
0FH	See Table 2-48
MSR_MS_COUNTER3	
0FH	See Table 2-48
MSR_MS_ESCRO	
0FH	See Table 2-48
MSR_MS_ESCR1	
0FH	See Table 2-48
MSR_MTRRCAP	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_OFFCORE_RSP_0	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
06_57H, 06_85H	See Table 2-46
MSR_OFFCORE_RSP_1	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_25H, 06_2CH	See Table 2-18
06_2FH	See Table 2-19
06_2AH, 06_2DH	See Table 2-20
06_57H, 06_85H	See Table 2-46

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_PCIE_PLL_RATIO	
06_3FH.....	See Table 2-32
MSR_PCU_PMON_BOX_CTL	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_BOX_FILTER	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_BOX_STATUS	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
MSR_PCU_PMON_CTR0	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_CTR1	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_CTR2	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_CTR3	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSELO	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL1	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL2	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PCU_PMON_EVNTSEL3	
06_2DH.....	See Table 2-24
06_3FH.....	See Table 2-33
MSR_PEBS_DATA_CFG	
06_7DH, 06_7EH.....	See Table 2-44
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

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_7AH.....	See Table 2-13
06_86H.....	See Table 2-14
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3EH.....	See Table 2-27
06_57H, 06_85H.....	See Table 2-46
0FH.....	See Table 2-48
MSR_PEBS_FRONTEND	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_PEBS_LD_LAT	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
MSR_PEBS_MATRIX_VERT	
0FH.....	See Table 2-48
MSR_PEBS_NUM_ALT	
06_2DH.....	See Table 2-23
MSR_PERF_CAPABILITIES	
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 2-15
MSR_PERF_GLOBAL_STATUS	
06_0FH, 06_17H.....	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
MSR_PERF_METRICS	
06_7DH, 06_7EH.....	See Table 2-44
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 2-20
MSR_PKG_C10_RESIDENCY	
06_5CH, 06_7AH.....	See Table 2-12
06_45H.....	See Table 2-30 and Table 2-31
06_4FH.....	See Table 2-38
MSR_PKG_C2_RESIDENCY	
06_27H.....	See Table 2-5
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 2-20
06_57H, 06_85H.....	See Table 2-46

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_PKG_C3_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
06_66H.....	See Table 2-42
06_57H, 06_85H.....	See Table 2-46
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, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
06_57H, 06_85H	See Table 2-46
MSR_PKG_C7_RESIDENCY	
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-15
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
06_57H, 06_85H	See Table 2-46
MSR_PKG_C8_RESIDENCY	
06_45H.....	See Table 2-31
06_4FH.....	See Table 2-38
MSR_PKG_C9_RESIDENCY	
06_45H.....	See Table 2-31
06_4FH.....	See Table 2-38
MSR_PKG_CST_CONFIG_CONTROL	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_4CH.....	See Table 2-11
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26
06_3CH, 06_45H, 06_46H	See Table 2-30
06_45H.....	See Table 2-31
06_3F.....	See Table 2-32
06_3DH.....	See Table 2-35
06_56H, 06_4FH	See Table 2-36
06_57H, 06_85H	See Table 2-46
MSR_PKG_ENERGY_STATUS	
06_37H, 06_4AH, 06_4CH, 06_5AH, 06_5DH	See Table 2-8
06_5CH, 06_7AH, 06_86H	See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3DH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
MSR_PKG_HDC_CONFIG	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_PKG_HDC_DEEP_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_PKG_HDC_SHALLOW_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_PKG_PERF_STATUS	
06_5CH, 06_7AH, 06_86H.....	See Table 2-12
06_2DH.....	See Table 2-23
06_3AH, 06_3EH.....	See Table 2-26
06_3CH, 06_3DH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH.....	See Table 2-29
06_57H, 06_85H.....	See Table 2-46
MSR_PKG_POWER_INFO	
06_4DH.....	See Table 2-10
06_5CH, 06_7AH, 06_86H.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3DH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
MSR_PKG_POWER_LIMIT	
06_37H, 06_4AH, 06_4CH, 06_5AH, 06_5DH.....	See Table 2-8
06_4DH.....	See Table 2-10
06_5CH, 06_7AH, 06_86H.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3DH, 06_3EH, 06_3FH, 06_45H, 06_46H, 06_47H, 06_4EH, 06_4FH, 06_55H, 06_56H, 06_5EH, 06_66H, 06_8EH, 06_9EH, 06_7DH, 06_7EH	See Table 2-20
06_57H, 06_85H.....	See Table 2-46
MSR_PKGC_IRTL1	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-29
MSR_PKGC_IRTL2	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-29
MSR_PKGC3_IRTL	
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_2DH.....	See Table 2-20
MSR_PKGC6_IRTL	
06_2AH, 06_2DH.....	See Table 2-20
MSR_PKGC7_IRTL	
06_2AH.....	See Table 2-21
MSR_PLATFORM_BRV	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 2-48
MSR_PLATFORM_ENERGY_COUNTER	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
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, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
MSR_PLATFORM_INFO	
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26
06_3CH, 06_45H, 06_46H.....	See Table 2-29 and Table 2-30
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-46
MSR_PLATFORM_POWER_LIMIT	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_PMG_IO_CAPTURE_BASE	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_4CH.....	See Table 2-11
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3AH.....	See Table 2-25
06_3EH.....	See Table 2-26
06_57H.....	See Table 2-46
MSR_PMH_ESCRO	
0FH.....	See Table 2-48
MSR_PMH_ESCR1	
0FH.....	See Table 2-48
MSR_PMON_GLOBAL_CONFIG	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
MSR_PMON_GLOBAL_CTL	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33
MSR_PMON_GLOBAL_STATUS	
06_3EH.....	See Table 2-28
06_3FH.....	See Table 2-33

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_POWER_CTL	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-15
06_2AH, 06_2DH	See Table 2-20
MSR_PPO_ENERGY_STATUS	
06_37H, 06_4AH, 06_5AH, 06_5DH	See Table 2-8
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-20
06_57H.....	See Table 2-46
MSR_PPO_POLICY	
06_2AH, 06_45H	See Table 2-21
MSR_PPO_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 2-20
06_57H.....	See Table 2-46
MSR_PP1_ENERGY_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_45H	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-30
MSR_PP1_POLICY	
06_2AH, 06_45H	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-30
MSR_PP1_POWER_LIMIT	
06_2AH, 06_45H	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-30
MSR_PPERF	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_PPIN	
06_3EH.....	See Table 2-26
06_56H, 06_4FH	See Table 2-36
MSR_PPIN_CTL	
06_3EH.....	See Table 2-26
06_56H, 06_4FH	See Table 2-36
MSR_PRMRR_BASE_0	
06_7DH, 06_7EH	See Table 2-44
MSR_PRMRR_PHYS_BASE	
06_8EH, 06_9EH	See Table 2-41
MSR_PRMRR_PHYS_MASK	
06_8EH, 06_9EH	See Table 2-41
MSR_PRMRR_VALID_CONFIG	
06_8EH, 06_9EH	See Table 2-41
MSR_RELOAD_FIXED_CTRx	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_86H	See Table 2-14
MSR_RELOAD_PMCx	
06_86H	See Table 2-14
MSR_RING_RATIO_LIMIT	
06_8EH, 06_9EH	See Table 2-41
MSR_RO_PMON_BOX_CTRL	
06_2EH	See Table 2-17
MSR_RO_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_RO_PMON_BOX_STATUS	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL0	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL1	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL2	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL3	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL4	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL5	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL6	
06_2EH	See Table 2-17
MSR_RO_PMON_CTRL7	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL0	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL1	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL2	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL3	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL4	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL5	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL6	
06_2EH	See Table 2-17
MSR_RO_PMON_EVTN_SEL7	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P0	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P1	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P2	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P3	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P4	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P5	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P6	
06_2EH.....	See Table 2-17
MSR_RO_PMON_IPERFO_P7	
06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P0	
06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P1	
06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P2	
06_2EH.....	See Table 2-17
MSR_RO_PMON_QLX_P3	
06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_R1_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR10	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR11	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR12	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR13	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR14	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR15	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR8	
06_2EH.....	See Table 2-17
MSR_R1_PMON_CTR9	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL10	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL11	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL12	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL13	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL14	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL15	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL8	
06_2EH.....	See Table 2-17
MSR_R1_PMON_EVTN_SEL9	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P10	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P11	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P12	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P13	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P14	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P15	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P8	
06_2EH.....	See Table 2-17
MSR_R1_PMON_IPERF1_P9	
06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P4	
06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P5	
06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P6	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
MSR_R1_PMON_QLX_P7	
06_2EH.....	See Table 2-17
MSR_RAPL_POWER_UNIT	
06_37H, 06_4AH, 06_5AH, 06_5DH.....	See Table 2-8
06_4DH.....	See Table 2-10
06_5CH, 06_7AH.....	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 2-20
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-46
MSR_RAT_ESCR0	
0FH.....	See Table 2-48
MSR_RAT_ESCR1	
0FH.....	See Table 2-48
MSR_RING_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H.....	See Table 2-30
MSR_SO_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_SO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_SO_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_SO_PMON_CTRL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_CTRL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_SO_PMON_EVTN_SEL1	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S0_PMON_EVNT_SEL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S0_PMON_EVNT_SEL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S0_PMON_MASK	
06_2EH.....	See Table 2-17
MSR_S0_PMON_MATCH	
06_2EH.....	See Table 2-17
MSR_S1_PMON_BOX_CTRL	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_S1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 2-17
MSR_S1_PMON_BOX_STATUS	
06_2EH.....	See Table 2-17
MSR_S1_PMON_CTR0	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTR1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTR2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_CTR3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SELO	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL1	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_EVNT_SEL2	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_S1_PMON_EVNT_SEL3	
06_2EH.....	See Table 2-17
06_3FH.....	See Table 2-33
MSR_S1_PMON_MASK	
06_2EH.....	See Table 2-17
MSR_S1_PMON_MATCH	
06_2EH.....	See Table 2-17
MSR_S2_PMON_BOX_CTL	
06_3FH.....	See Table 2-33
MSR_S2_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_S2_PMON_CTRL0	
06_3FH.....	See Table 2-33
MSR_S2_PMON_CTRL1	
06_3FH.....	See Table 2-33
MSR_S2_PMON_CTRL2	
06_3FH.....	See Table 2-33
MSR_S2_PMON_CTRL3	
06_3FH.....	See Table 2-33
MSR_S2_PMON_EVTSELO	
06_3FH.....	See Table 2-33
MSR_S2_PMON_EVTSEL1	
06_3FH.....	See Table 2-33
MSR_S2_PMON_EVTSEL2	
06_3FH.....	See Table 2-33
MSR_S2_PMON_EVTSEL3	
06_3FH.....	See Table 2-33
MSR_S3_PMON_BOX_CTL	
06_3FH.....	See Table 2-33
MSR_S3_PMON_BOX_FILTER	
06_3FH.....	See Table 2-33
MSR_S3_PMON_CTRL0	
06_3FH.....	See Table 2-33
MSR_S3_PMON_CTRL1	
06_3FH.....	See Table 2-33
MSR_S3_PMON_CTRL2	
06_3FH.....	See Table 2-33
MSR_S3_PMON_CTRL3	
06_3FH.....	See Table 2-33
MSR_S3_PMON_EVTSELO	
06_3FH.....	See Table 2-33
MSR_S3_PMON_EVTSEL1	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH.....	See Table 2-33
MSR_S3_PMON_EVTSEL2	
06_3FH.....	See Table 2-33
MSR_S3_PMON_EVTSEL3	
06_3FH.....	See Table 2-33
MSR_SAAT_ESCR0	
0FH.....	See Table 2-48
MSR_SAAT_ESCR1	
0FH.....	See Table 2-48
MSR_SGXOWNEREPOCH0	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_SGXOWNEREPOCH1	
06_5CH, 06_7AH.....	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH.....	See Table 2-39
MSR_SMI_COUNT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_57H.....	See Table 2-46
MSR_SMM_BLOCKED	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-30
MSR_SMM_DELAYED	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-30
MSR_SMM_FEATURE_CONTROL	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-30
MSR_SMM_MCA_CAP	
06_5CH, 06_7AH.....	See Table 2-12
06_3CH, 06_45H, 06_46H.....	See Table 2-30
06_3FH.....	See Table 2-32
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-46
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 2-48
MSR_TBPU_ESCR0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 2-48
MSR_TBPU_ESCR1	
0FH.....	See Table 2-48
MSR_TC_ESCR0	
0FH.....	See Table 2-48
MSR_TC_ESCR1	
0FH.....	See Table 2-48
MSR_TC_PRECISE_EVENT	
0FH.....	See Table 2-48
MSR_TEMPERATURE_TARGET	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
06_2AH, 06_2DH.....	See Table 2-20
06_3EH.....	See Table 2-26
06_56H, 06_4FH.....	See Table 2-36
06_57H.....	See Table 2-46
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 2-48
06_0EH.....	See Table 2-51
06_09H.....	See Table 2-52
MSR_THREAD_ID_INFO	
06_3FH.....	See Table 2-32
MSR_TRACE_HUB_STH ACPIBAR_BASE	
06_8EH, 06_9EH.....	See Table 2-41
MSR_TURBO_ACTIVATION_RATIO	
06_5CH, 06_7AH.....	See Table 2-12
06_3AH.....	See Table 2-25
06_3CH, 06_45H, 06_46H.....	See Table 2-29
06_57H.....	See Table 2-46
MSR_TURBO_GROUP_CORECNT	
06_5CH, 06_7AH.....	See Table 2-12
MSR_TURBO_POWER_CURRENT_LIMIT	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 2-15
MSR_TURBO_RATIO_LIMIT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.....	See Table 2-6
06_4DH.....	See Table 2-10
06_5CH, 06_7AH.....	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH.....	See Table 2-15
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH.....	See Table 2-16
06_2EH.....	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_25H, 06_2CH	See Table 2-18
06_2FH	See Table 2-19
06_2AH, 06_45H	See Table 2-21
06_2DH	See Table 2-23
06_3EH	See Table 2-26 and Table 2-27
06_3CH, 06_45H, 06_46H	See Table 2-30
06_3FH	See Table 2-32
06_3DH	See Table 2-35
06_56H, 06_4FH	See Table 2-36
06_55H	See Table 2-45
06_57H	See Table 2-46
MSR_TURBO_RATIO_LIMIT1	
06_3EH	See Table 2-26 and Table 2-27
06_3FH	See Table 2-32
06_56H, 06_4FH	See Table 2-36
MSR_TURBO_RATIO_LIMIT2	
06_3FH	See Table 2-32
MSR_TURBO_RATIO_LIMIT3	
06_56H	See Table 2-37
06_4FH	See Table 2-38
MSR_TURBO_RATIO_LIMIT_CORES	
06_55H	See Table 2-45
MSR_U_PMON_BOX_STATUS	
06_3EH	See Table 2-28
06_3FH	See Table 2-33
MSR_U_PMON_CTR	
06_2EH	See Table 2-17
MSR_U_PMON_CTR0	
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_U_PMON_CTR1	
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_U_PMON_EVNT_SEL	
06_2EH	See Table 2-17
MSR_U_PMON_EVNTSELO	
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_U_PMON_EVNTSEL1	
06_2DH	See Table 2-24

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-33
MSR_U_PMON_GLOBAL_CTRL	
06_2EH	See Table 2-17
MSR_U_PMON_GLOBAL_OVF_CTRL	
06_2EH	See Table 2-17
MSR_U_PMON_GLOBAL_STATUS	
06_2EH	See Table 2-17
MSR_U_PMON_UCLK_FIXED_CTL	
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_U_PMON_UCLK_FIXED_CTR	
06_2DH	See Table 2-24
06_3FH	See Table 2-33
MSR_U2L_ESCR0	
0FH	See Table 2-48
MSR_U2L_ESCR1	
0FH	See Table 2-48
MSR_UNC_ARB_PERFCTRO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_ARB_PERFCTR1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_ARB_PERFEVTSELO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_ARB_PERFEVTSEL1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_0_PERFCTRO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_0_PERFCTR1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_0_PERFCTR2	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH	See Table 2-22
MSR_UNC_CBO_0_PERFCTR3	
06_2AH	See Table 2-22
MSR_UNC_CBO_0_PERFEVTSELO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_0_PERFEVTSEL1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_0_PERFEVTSEL2	
06_2AH	See Table 2-22
MSR_UNC_CBO_0_PERFEVTSEL3	
06_2AH	See Table 2-22
MSR_UNC_CBO_0_UNIT_STATUS	
06_2AH	See Table 2-22
MSR_UNC_CBO_1_PERFCTR0	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_1_PERFCTR1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_1_PERFCTR2	
06_2AH	See Table 2-22
MSR_UNC_CBO_1_PERFCTR3	
06_2AH	See Table 2-22
MSR_UNC_CBO_1_PERFEVTSELO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_1_PERFEVTSEL1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_1_PERFEVTSEL2	
06_2AH	See Table 2-22
MSR_UNC_CBO_1_PERFEVTSEL3	
06_2AH	See Table 2-22
MSR_UNC_CBO_1_UNIT_STATUS	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH	See Table 2-22
MSR_UNC_CBO_2_PERFCTR0	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_2_PERFCTR1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_2_PERFCTR2	
06_2AH	See Table 2-22
MSR_UNC_CBO_2_PERFCTR3	
06_2AH	See Table 2-22
MSR_UNC_CBO_2_PERFEVTSELO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_2_PERFEVTSEL1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_2_PERFEVTSEL2	
06_2AH	See Table 2-22
MSR_UNC_CBO_2_PERFEVTSEL3	
06_2AH	See Table 2-22
MSR_UNC_CBO_2_UNIT_STATUS	
06_2AH	See Table 2-22
MSR_UNC_CBO_3_PERFCTR0	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_3_PERFCTR1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_3_PERFCTR2	
06_2AH	See Table 2-22
MSR_UNC_CBO_3_PERFCTR3	
06_2AH	See Table 2-22
MSR_UNC_CBO_3_PERFEVTSELO	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_3_PERFEVTSEL1	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_CBO_3_PERFEVTSEL2	
06_2AH	See Table 2-22
MSR_UNC_CBO_3_PERFEVTSEL3	
06_2AH	See Table 2-22
MSR_UNC_CBO_3_UNIT_STATUS	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFCTR0	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFCTR1	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFCTR2	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFCTR3	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSELO	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL1	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL2	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_PERFEVTSEL3	
06_2AH	See Table 2-22
MSR_UNC_CBO_4_UNIT_STATUS	
06_2AH	See Table 2-22
MSR_UNC_CBO_CONFIG	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_PERF_FIXED_CTR	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_PERF_FIXED_CTRL	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_PERF_GLOBAL_CTRL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNC_PERF_GLOBAL_STATUS	
06_2AH	See Table 2-22
06_3CH, 06_45H, 06_46H	See Table 2-30
06_4EH, 06_5EH	See Table 2-40
MSR_UNCORE_ADDR_OPCODE_MATCH	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_FIXED_CTR_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_FIXED_CTR0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERF_GLOBAL_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL3	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PERFEVTSEL7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC3	

MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
06_2EH	See Table 2-17
MSR_UNCORE_PMC6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PMC7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-16
MSR_UNCORE_PRMRR_BASE	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_UNCORE_PRMRR_MASK	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MSR_UNCORE_PRMRR_PHYS_BASE	
06_8EH, 06_9EH	See Table 2-41
MSR_UNCORE_PRMRR_PHYS_MASK	
06_8EH, 06_9EH	See Table 2-41
MSR_W_PMON_BOX_CTRL	
06_2EH	See Table 2-17
MSR_W_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-17
MSR_W_PMON_BOX_STATUS	
06_2EH	See Table 2-17
MSR_W_PMON_CTR0	
06_2EH	See Table 2-17
MSR_W_PMON_CTR1	
06_2EH	See Table 2-17
MSR_W_PMON_CTR2	
06_2EH	See Table 2-17
MSR_W_PMON_CTR3	
06_2EH	See Table 2-17
MSR_W_PMON_EVNT_SELO	
06_2EH	See Table 2-17
MSR_W_PMON_EVNT_SEL1	
06_2EH	See Table 2-17
MSR_W_PMON_EVNT_SEL2	
06_2EH	See Table 2-17
MSR_W_PMON_EVNT_SEL3	
06_2EH	See Table 2-17
MSR_W_PMON_FIXED_CTR	
06_2EH	See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_W_PMON_FIXED_CTR_CTL	
06_2EH	See Table 2-17
MSR_WEIGHTED_CORE_CO	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H, 06_7DH, 06_7EH	See Table 2-39
MTRRfix16K_80000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix16K_A0000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_C0000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_C8000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_D0000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_D8000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_E0000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_E8000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_F0000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix4K_F8000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRfix64K_00000	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase0	
06_0EH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase1	
06_0EH	See Table 2-51

MSR Name and CPUID DisplayFamily_DisplayModel	Location
P6 Family	See Table 2-53
MTRRphysBase2	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase3	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase4	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase5	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase6	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysBase7	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask0	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask1	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask2	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask3	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask4	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask5	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask6	
06_OEH	See Table 2-51
P6 Family	See Table 2-53
MTRRphysMask7	
06_OEH	See Table 2-51

MSR Name and CPUID DisplayFamily_DisplayModel

Location

	P6 Family	See Table 2-53
■	MSR_TEST_CTRL	
■	06_86H	See Table 2-14
■	06_7DH, 06_7EH	See Table 2-44
	P6 Family	See Table 2-53

