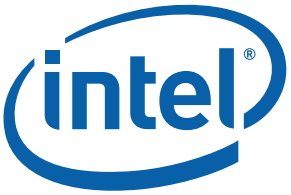


Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

December 2017

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

Document Number: 252046-057

Contents

Revision History

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

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

Revision	Description	Date
-048	• Removed Documentation Changes 1-19 • Add Documentation Changes 1-33	September 2015
-049	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-33	December 2015
-050	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-9	April 2016
-051	• Removed Documentation Changes 1-9 • Add Documentation Changes 1-20	June 2016
-052	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-22	September 2016
-053	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-26	December 2016
-054	• Removed Documentation Changes 1-26 • Add Documentation Changes 1-20	March 2017
-055	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-28	July 2017
-056	• Removed Documentation Changes 1-28 • Add Documentation Changes 1-18	October 2017
-057	• Removed Documentation Changes 1-18 • Add Documentation Changes 1-29	December 2017

§

Preface

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

Affected Documents

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

Nomenclature

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

Summary Tables of Changes

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

Codes Used in Summary Tables

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

Documentation Changes(Sheet 1 of 2)

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 3, Volume 1
2	Updates to Chapter 4, Volume 1
3	Updates to Chapter 5, Volume 1
4	Updates to Chapter 8, Volume 1
5	Updates to Chapter 11, Volume 1
6	Updates to Chapter 19, Volume 1
7	Updates to Appendix D, Volume 1
8	Updates to Appendix E, Volume 1
9	Updates to Chapter 2, Volume 2A
10	Updates to Chapter 3, Volume 2A
11	Updates to Chapter 4, Volume 2B
12	Updates to Chapter 5, Volume 2C
13	Updates to Chapter 7, Volume 2D
14	Updates to Chapter 8, Volume 3A
15	Updates to Chapter 9, Volume 3A
16	Updates to Chapter 10, Volume 3A
17	Updates to Chapter 17, Volume 3B
18	Updates to Chapter 18, Volume 3B
19	Updates to Chapter 19, Volume 3B
20	Updates to Chapter 24, Volume 3B
21	Updates to Chapter 28, Volume 3C
22	Updates to Chapter 35, Volume 3C
23	Updates to Chapter 36, Volume 3D
24	Updates to Chapter 37, Volume 3D
25	Updates to Chapter 38, Volume 3D
26	Updates to Chapter 40, Volume 3D

Documentation Changes(Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
27	Updates to Chapter 41, Volume 3D
28	Updates to Appendix A, Volume 3D
29	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 3, Volume 1

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

Change to this chapter: Minor correction in Section 3.4.2 "Segment Registers" and Section 3.7.4 "Specifying a Segment Selector".

CHAPTER 3
BASIC EXECUTION ENVIRONMENT

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.

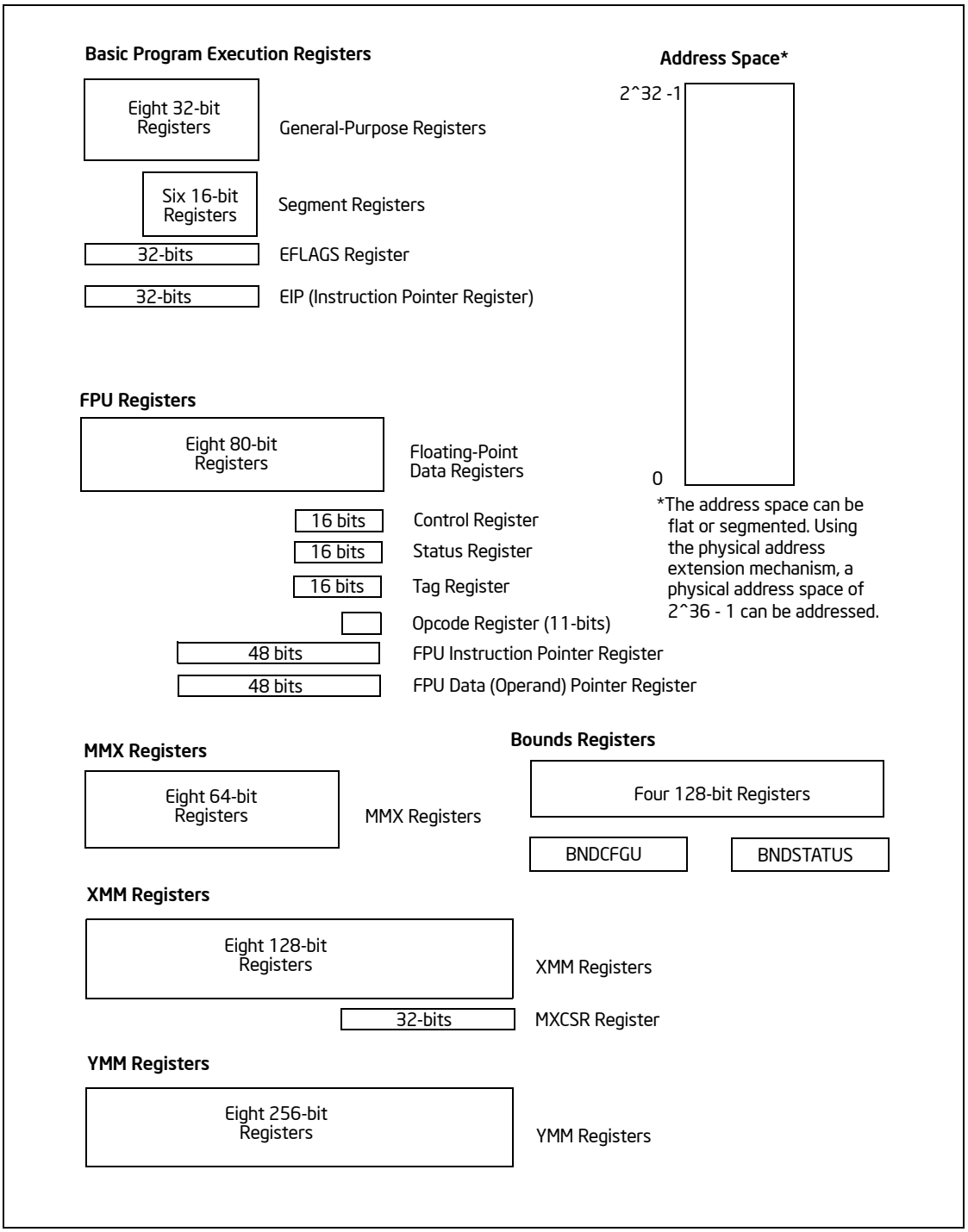


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

- **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 18, "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*.
- **Machine specific registers (MSRs)** — The processor provides a variety of machine 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^{46} 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.

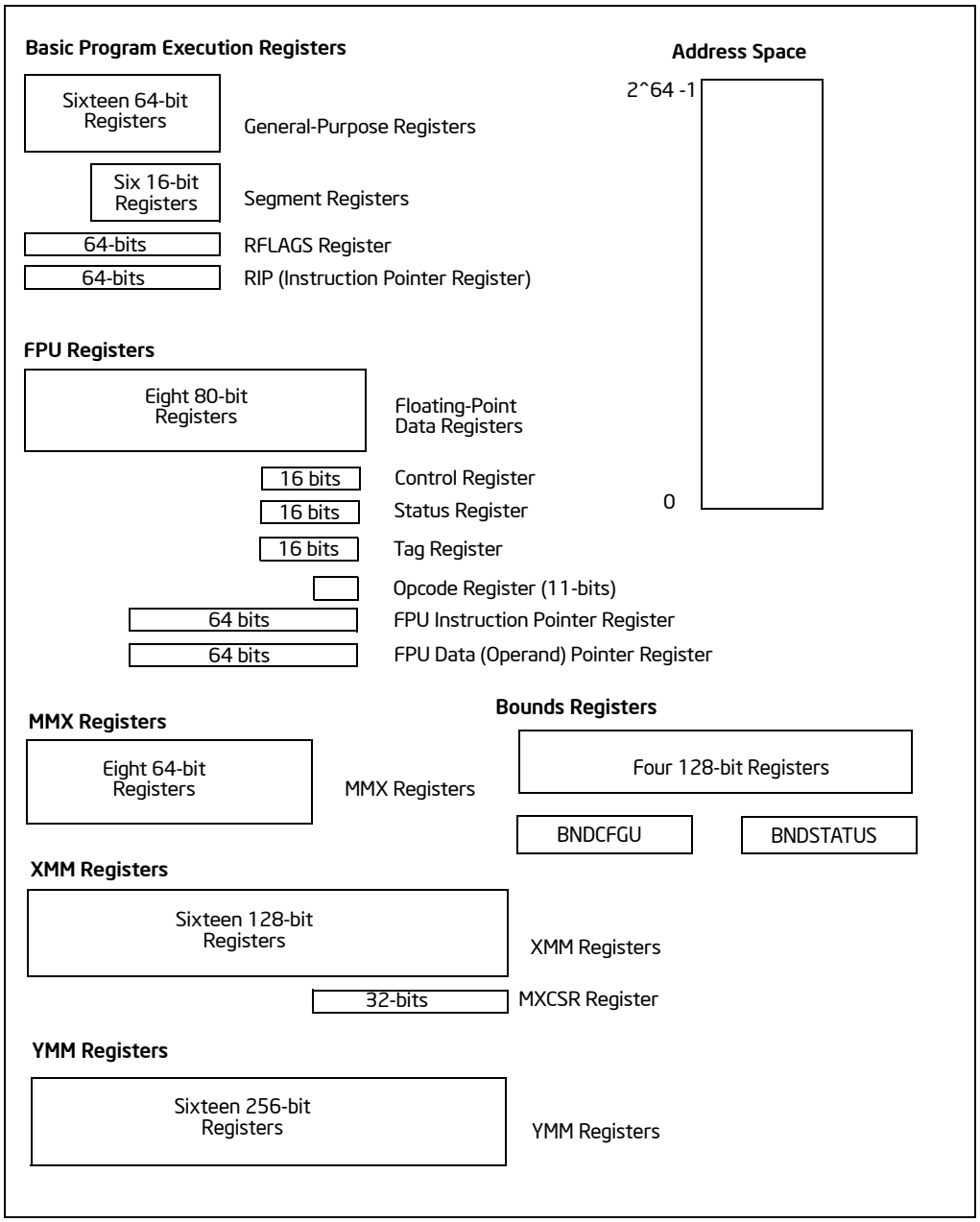


Figure 3-2. 64-Bit Mode Execution Environment

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 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.*

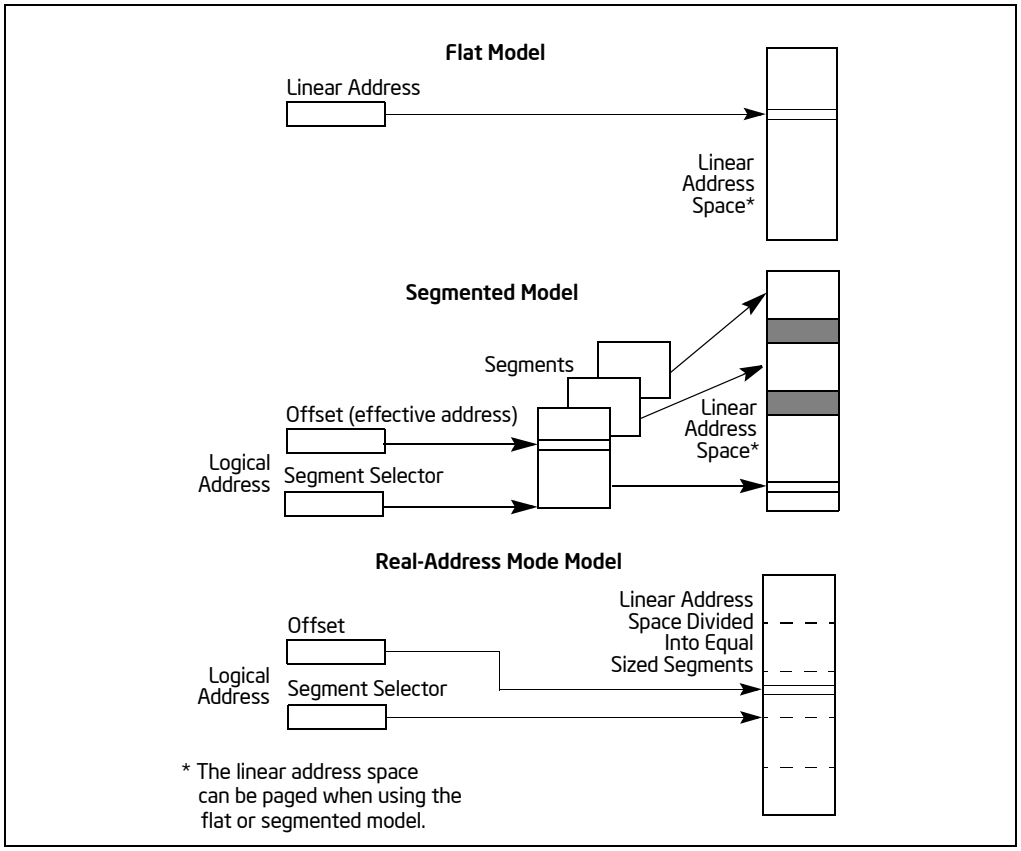


Figure 3-3. Three Memory Management Models

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 000FFFFFH (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 programing (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.

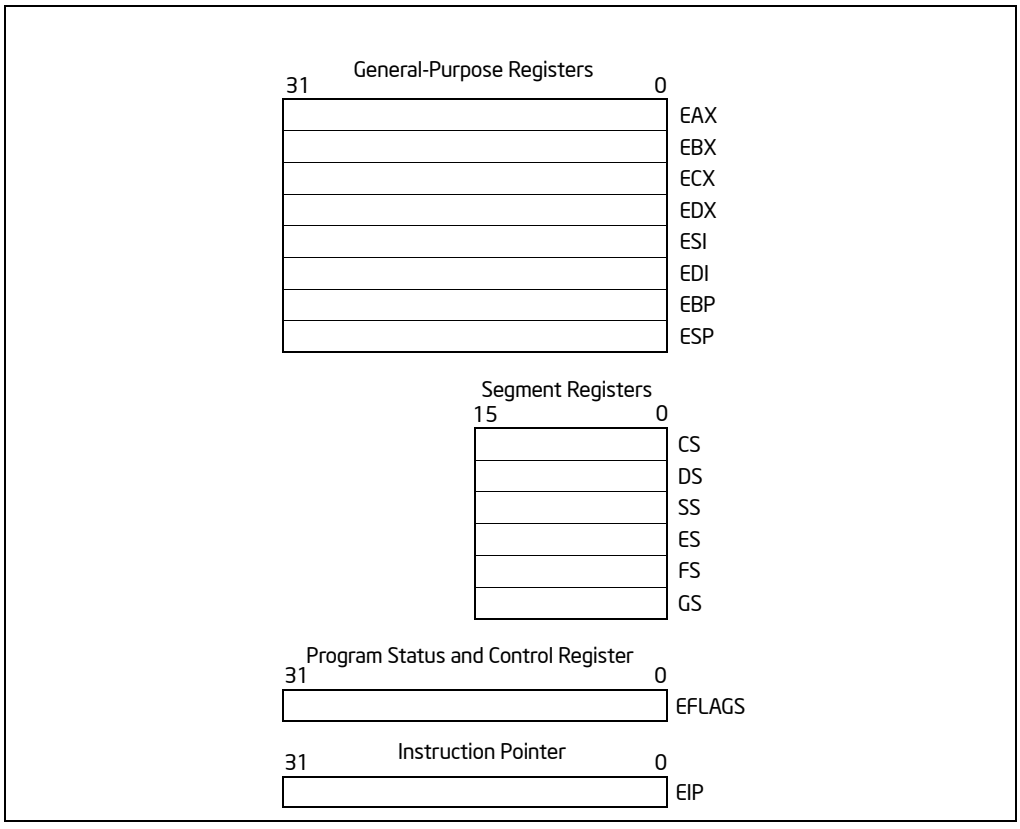


Figure 3-4. General System and Application Programming Registers

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).

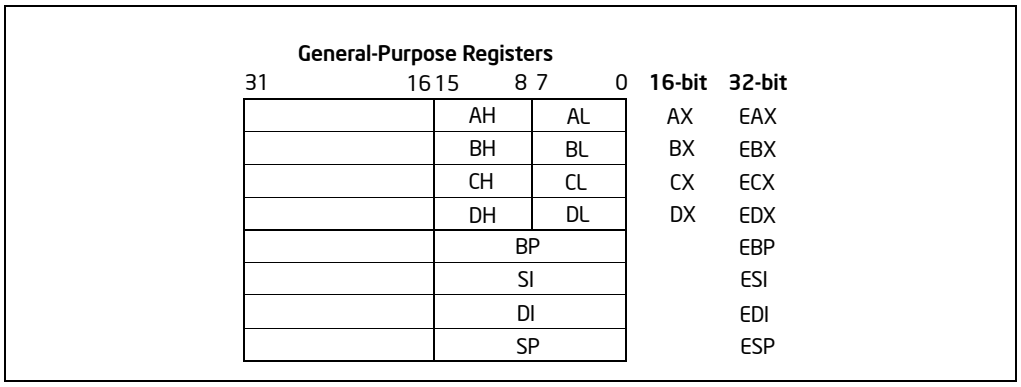


Figure 3-5. Alternate General-Purpose Register Names

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, R8L - R15L
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.

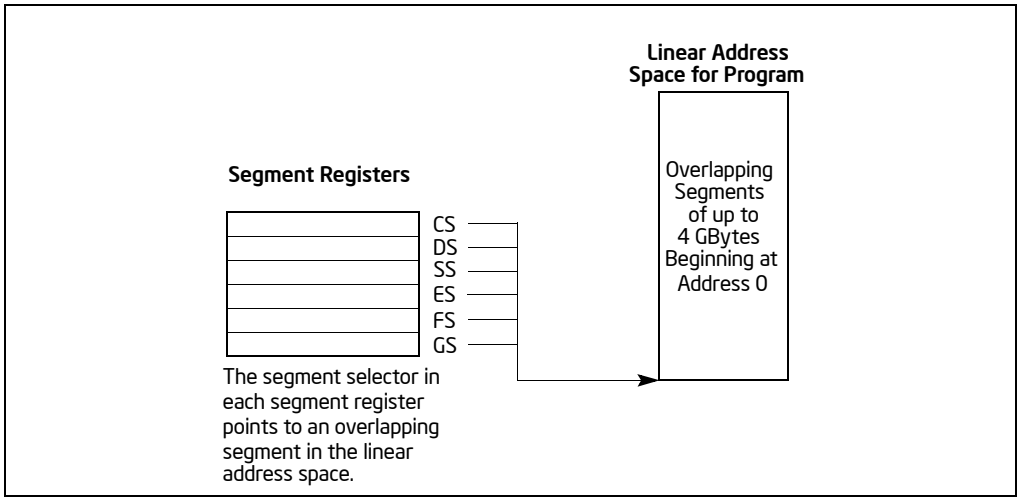


Figure 3-6. Use of Segment Registers for Flat Memory Model

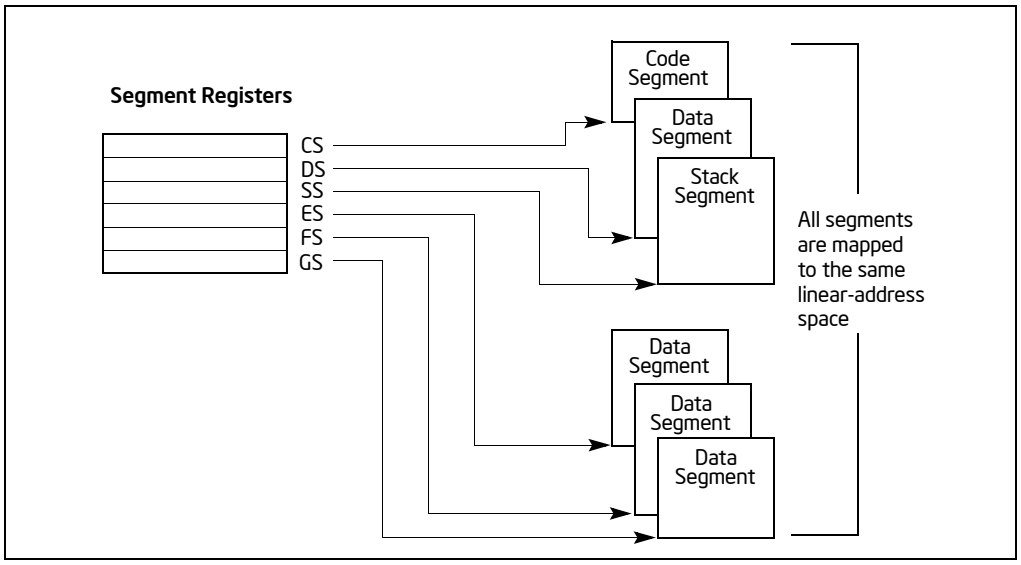


Figure 3-7. Use of Segment Registers in Segmented Memory Model

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.

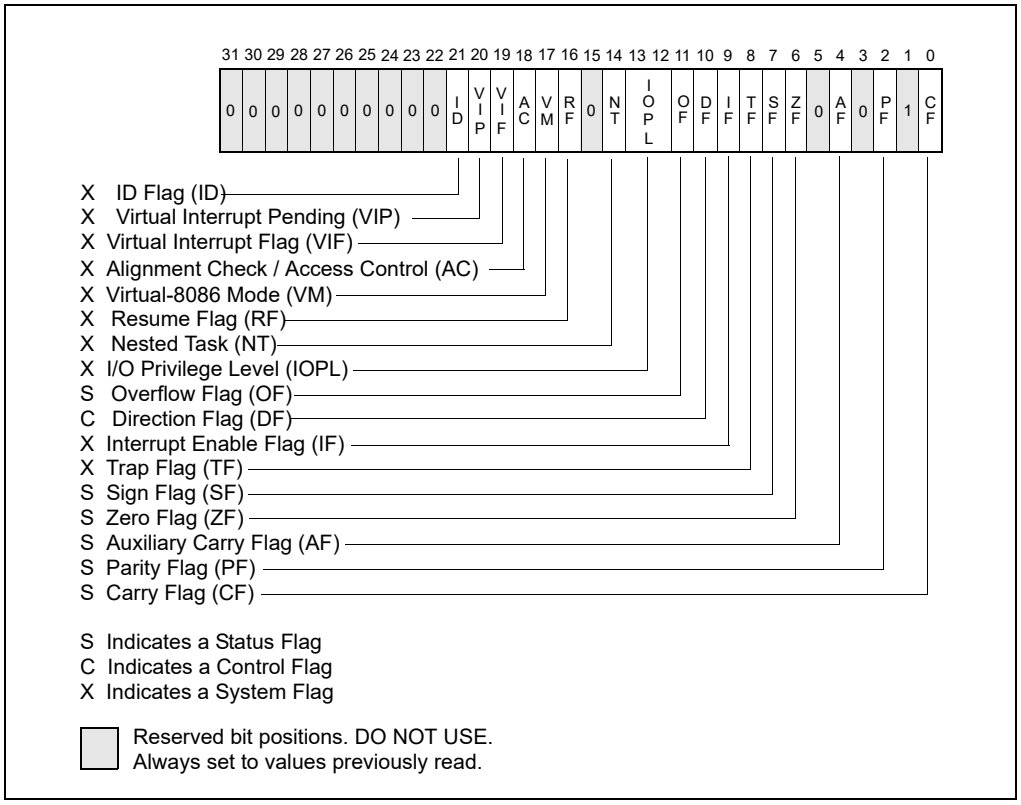


Figure 3-8. EFLAGS Register

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 J*cc* (jump on condition code *cc*), SET*cc* (byte set on condition code *cc*), LOOP*cc*, and CMOV*cc* (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, J*cc*, 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, J*cc*, 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 J*cc* 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 R8L-R15L 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).

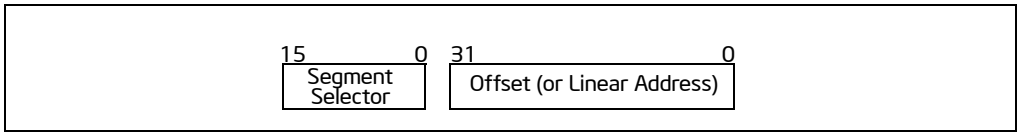


Figure 3-9. Memory Operand Address

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).

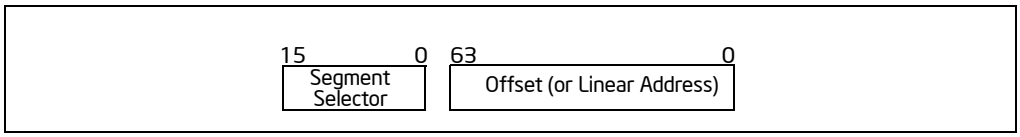


Figure 3-10. Memory Operand Address in 64-Bit Mode

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 (2s 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.

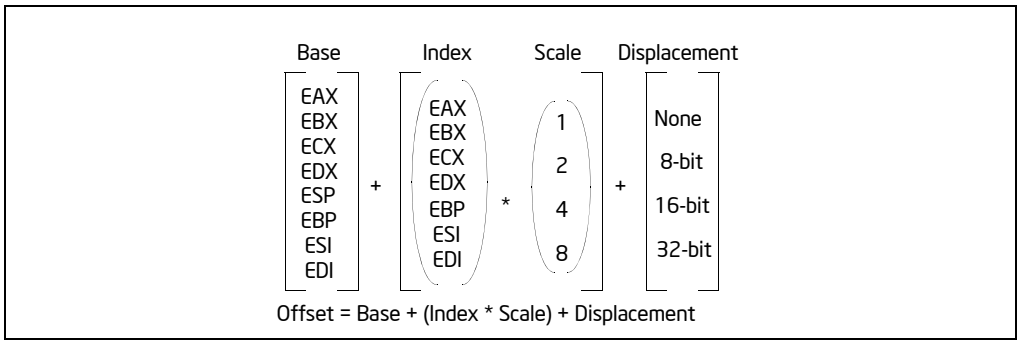


Figure 3-11. Offset (or Effective Address) Computation

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 18, "Input/Output," for more information about I/O port addressing.

2. Updates to Chapter 4, Volume 1

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Change to this chapter: Additions to the Section 4.8.3.5 "Operating on SNaNs and QNaNs". Small update to Section 4.9.1.1 "Invalid Operation Exception (#I)".

CHAPTER 4
DATA TYPES

This chapter introduces data types defined for the Intel 64 and IA-32 architectures. A section at the end of this chapter describes the real-number and floating-point concepts used in x87 FPU, SSE, SSE2, SSE3, SSSE3, SSE4 and Intel AVX extensions.

4.1 FUNDAMENTAL DATA TYPES

The fundamental data types are bytes, words, doublewords, quadwords, and double quadwords (see Figure 4-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), a quadword is 8 bytes (64 bits), and a double quadword is 16 bytes (128 bits). A subset of the IA-32 architecture instructions operates on these fundamental data types without any additional operand typing.

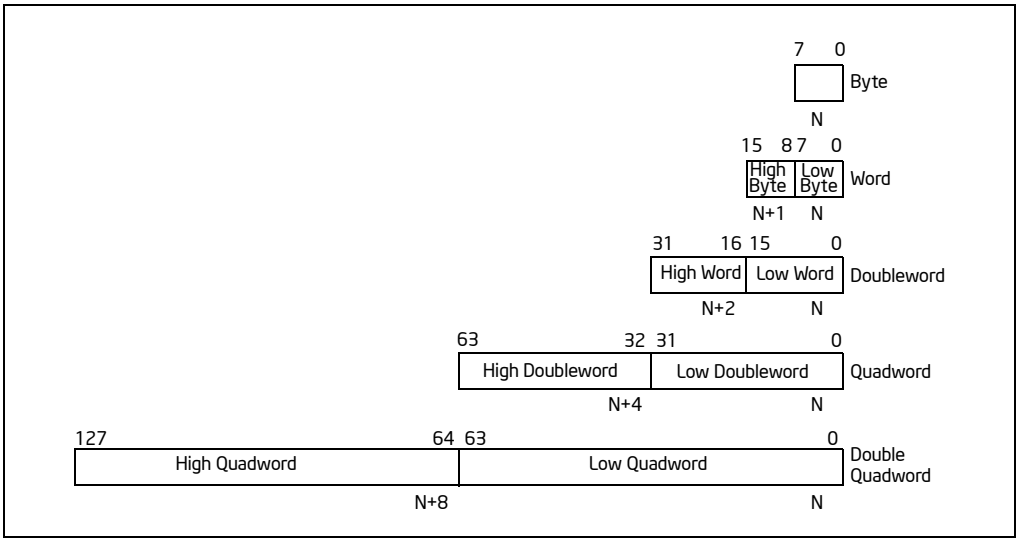


Figure 4-1. Fundamental Data Types

The quadword data type was introduced into the IA-32 architecture in the Intel486 processor; the double quadword data type was introduced in the Pentium III processor with the SSE extensions.

Figure 4-2 shows the byte order of each of the fundamental data types when referenced as operands in memory. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand.

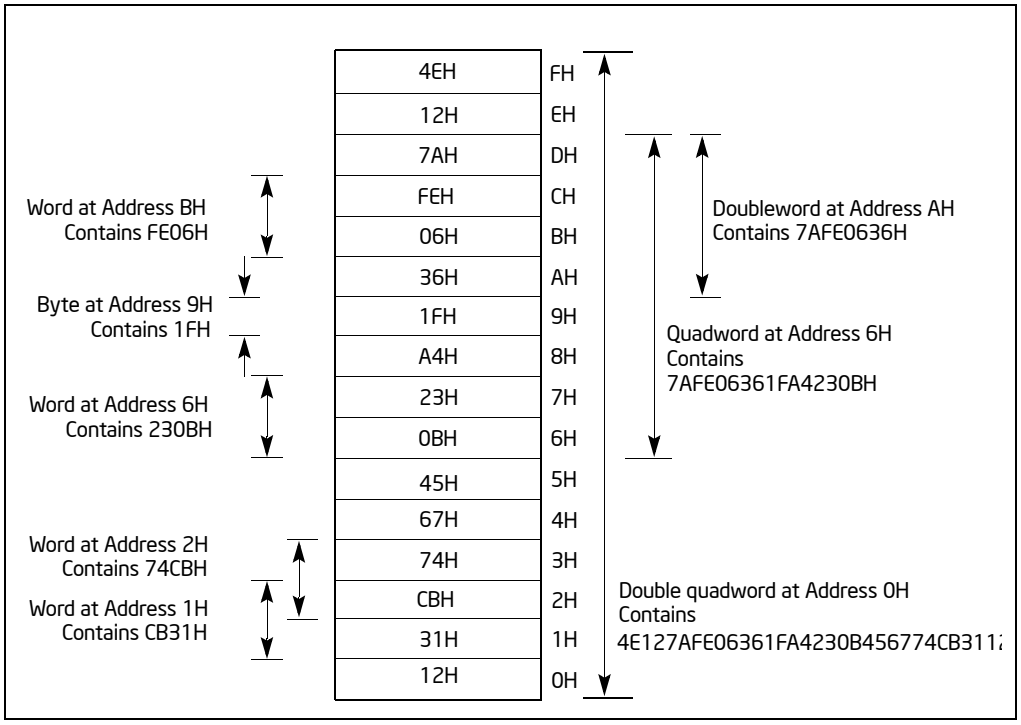


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

Some instructions that operate on double quadwords require memory operands to be aligned on a natural boundary. These instructions generate a general-protection exception (#GP) if an unaligned operand is specified. A natural boundary for a double quadword is any address evenly divisible by 16. Other instructions that operate on double quadwords permit unaligned access (without generating a general-protection exception). However, additional memory bus cycles are required to access unaligned data from memory.

4.2 NUMERIC DATA TYPES

Although bytes, words, and doublewords are fundamental data types, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (signed and unsigned integers, and floating-point numbers). Single-precision (32-bit) floating-point and double-precision (64-bit) floating-point data types are supported across all generations of SSE extensions and Intel AVX extensions. Half-precision (16-bit) floating-point data type is supported only with F16C extensions (VCVTPH2PS, VCVTPS2PH). See Figure 4-3.

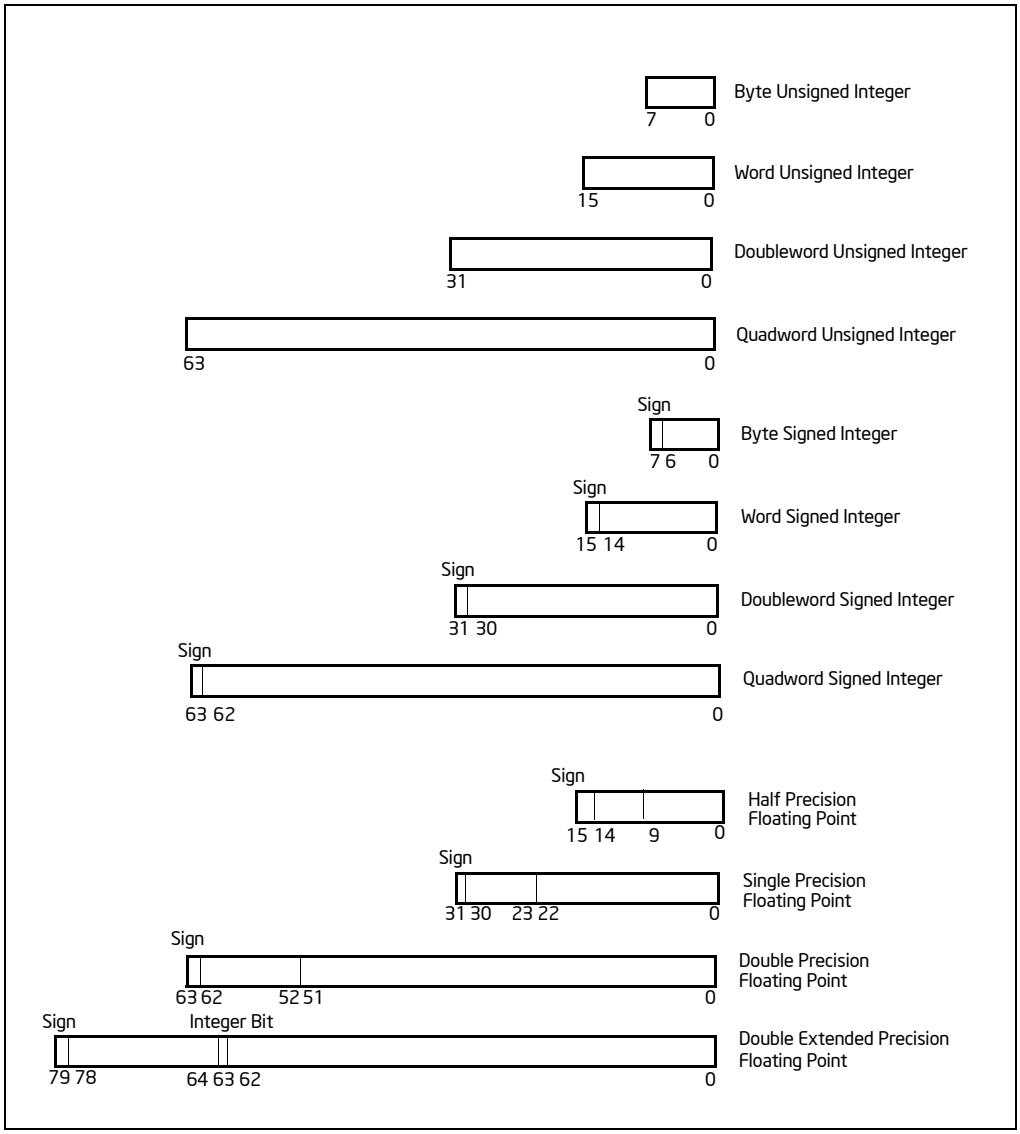


Figure 4-3. Numeric Data Types

4.2.1 Integers

The Intel 64 and IA-32 architectures define two types of integers: unsigned and signed. Unsigned integers are ordinary binary values ranging from 0 to the maximum positive number that can be encoded in the selected operand size. Signed integers are two's complement binary values that can be used to represent both positive and negative integer values.

Some integer instructions (such as the ADD, SUB, PADDB, and PSUBB instructions) operate on either unsigned or signed integer operands. Other integer instructions (such as IMUL, MUL, IDIV, DIV, FIADD, and FISUB) operate on only one integer type.

The following sections describe the encodings and ranges of the two types of integers.

4.2.1.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, word, doubleword, and quadword. Their values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer, from 0

to $2^{32} - 1$ for an unsigned doubleword integer, and from 0 to $2^{64} - 1$ for an unsigned quadword integer. Unsigned integers are sometimes referred to as **ordinals**.

4.2.1.2 Signed Integers

Signed integers are signed binary numbers held in a byte, word, doubleword, or quadword. All operations on signed integers assume a two's complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, bit 31 in a doubleword integer, and bit 63 in a quadword integer (see the signed integer encodings in Table 4-1).

Table 4-1. Signed Integer Encodings

Class		Two's Complement Encoding	
		Sign	
Positive	Largest	0	11..11
	.	.	.
	.	.	.
	Smallest	0	00..01
Zero		0	00..00
Negative	Smallest	1	11..11
	.	.	.
	.	.	.
	Largest	1	00..00
Integer indefinite		1	00..00
		Signed Byte Integer:	← 7 bits →
		Signed Word Integer:	← 15 bits →
		Signed Doubleword Integer:	← 31 bits →
		Signed Quadword Integer:	← 63 bits →

The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from −128 to +127 for a byte integer, from −32,768 to +32,767 for a word integer, from -2^{31} to $+2^{31} - 1$ for a doubleword integer, and from -2^{63} to $+2^{63} - 1$ for a quadword integer.

When storing integer values in memory, word integers are stored in 2 consecutive bytes; doubleword integers are stored in 4 consecutive bytes; and quadword integers are stored in 8 consecutive bytes.

The integer indefinite is a special value that is sometimes returned by the x87 FPU when operating on integer values. For more information, see Section 8.2.1, "Indefinites."

4.2.2 Floating-Point Data Types

The IA-32 architecture defines and operates on three floating-point data types: single-precision floating-point, double-precision floating-point, and double-extended precision floating-point (see Figure 4-3). The data formats for these data types correspond directly to formats specified in the IEEE Standard 754 for Binary Floating-Point Arithmetic.

Half-precision (16-bit) floating-point data type is supported only for conversion operation with single-precision floating data using F16C extensions (VCVTPH2PS, VCVTPS2PH).

Table 4-2 gives the length, precision, and approximate normalized range that can be represented by each of these data types. Denormal values are also supported in each of these types.

Table 4-2. Length, Precision, and Range of Floating-Point Data Types

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^{4}
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Double Extended Precision	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}

NOTE

Section 4.8, "Real Numbers and Floating-Point Formats," gives an overview of the IEEE Standard 754 floating-point formats and defines the terms integer bit, QNaN, SNaN, and denormal value.

Table 4-3 shows the floating-point encodings for zeros, denormalized finite numbers, normalized finite numbers, infinites, and NaNs for each of the three floating-point data types. It also gives the format for the QNaN floating-point indefinite value. (See Section 4.8.3.7, "QNaN Floating-Point Indefinite," for a discussion of the use of the QNaN floating-point indefinite value.)

For the single-precision and double-precision formats, only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the double extended-precision format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers.

Table 4-3. Floating-Point Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	–Zero	1	00..00	0	00..00
	–Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	–Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-∞	1	11..11	1	00..00

Table 4-3. Floating-Point Number and NaN Encodings (Contd.)

NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	QNaN Floating-Point Indefinite	1	11..11	1	10..00
	Half-Precision Single-Precision: Double-Precision: Double Extended-Precision:		← 5Bits → ← 8 Bits → ← 11 Bits → ← 15 Bits →		← 10 Bits → ← 23 Bits → ← 52 Bits → ← 63 Bits →

NOTES:

1. Integer bit is implied and not stored for single-precision and double-precision formats.
2. The fraction for SNaN encodings must be non-zero with the most-significant bit 0.

The exponent of each floating-point data type is encoded in biased format; see Section 4.8.2.2, "Biased Exponent." The biasing constant is 15 for the half-precision format, 127 for the single-precision format, 1023 for the double-precision format, and 16,383 for the double extended-precision format.

When storing floating-point values in memory, half-precision values are stored in 2 consecutive bytes in memory; single-precision values are stored in 4 consecutive bytes in memory; double-precision values are stored in 8 consecutive bytes; and double extended-precision values are stored in 10 consecutive bytes.

The single-precision and double-precision floating-point data types are operated on by x87 FPU, and SSE/SSE2/SSE3/SSE4.1 and Intel AVX instructions. The double-extended-precision floating-point format is only operated on by the x87 FPU. See Section 11.6.8, "Compatibility of SIMD and x87 FPU Floating-Point Data Types," for a discussion of the compatibility of single-precision and double-precision floating-point data types between the x87 FPU and SSE/SSE2/SSE3 extensions.

4.3 POINTER DATA TYPES

Pointers are addresses of locations in memory.

In non-64-bit modes, the architecture defines two types of pointers: a **near pointer** and a **far pointer**. A near pointer is a 32-bit (or 16-bit) offset (also called an **effective address**) within a segment. Near pointers are used for all memory references in a flat memory model or for references in a segmented model where the identity of the segment being accessed is implied.

A far pointer is a logical address, consisting of a 16-bit segment selector and a 32-bit (or 16-bit) offset. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly. Near and far pointers with 32-bit offsets are shown in Figure 4-4.

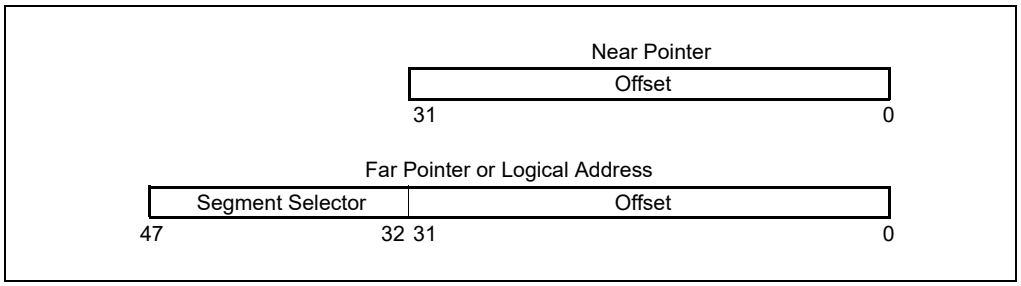


Figure 4-4. Pointer Data Types

4.3.1 Pointer Data Types in 64-Bit Mode

In 64-bit mode (a sub-mode of IA-32e mode), a **near pointer** is 64 bits. This equates to an effective address. **Far pointers** in 64-bit mode can be one of three forms:

- 16-bit segment selector, 16-bit offset if the operand size is 32 bits
- 16-bit segment selector, 32-bit offset if the operand size is 32 bits
- 16-bit segment selector, 64-bit offset if the operand size is 64 bits

See Figure 4-5.

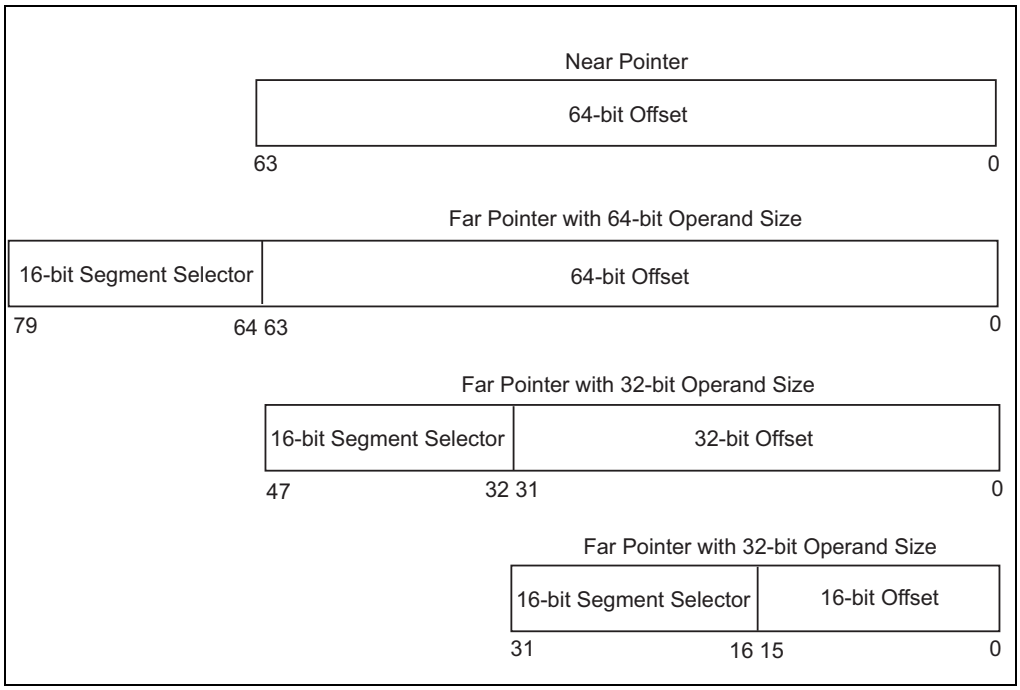


Figure 4-5. Pointers in 64-Bit Mode

4.4 BIT FIELD DATA TYPE

A **bit field** (see Figure 4-6) is a contiguous sequence of bits. It can begin at any bit position of any byte in memory and can contain up to 32 bits.

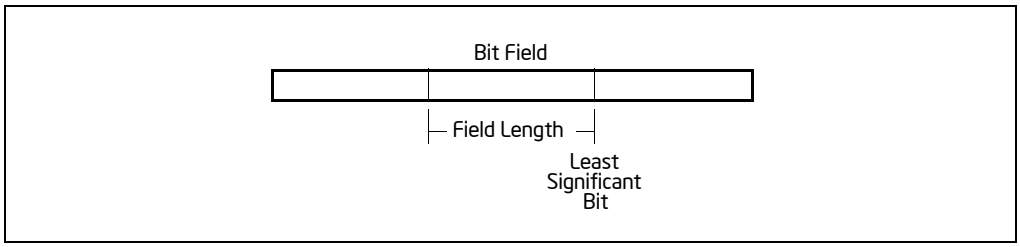


Figure 4-6. Bit Field Data Type

4.5 STRING DATA TYPES

Strings are continuous sequences of bits, bytes, words, or doublewords. A **bit string** can begin at any bit position of any byte and can contain up to $2^{32} - 1$ bits. A **byte string** can contain bytes, words, or doublewords and can range from zero to $2^{32} - 1$ bytes (4 GBytes).

4.6 PACKED SIMD DATA TYPES

Intel 64 and IA-32 architectures define and operate on a set of 64-bit and 128-bit packed data type for use in SIMD operations. These data types consist of fundamental data types (packed bytes, words, doublewords, and quadwords) and numeric interpretations of fundamental types for use in packed integer and packed floating-point operations.

4.6.1 64-Bit SIMD Packed Data Types

The 64-bit packed SIMD data types were introduced into the IA-32 architecture in the Intel MMX technology. They are operated on in MMX registers. The fundamental 64-bit packed data types are packed bytes, packed words, and packed doublewords (see Figure 4-7). When performing numeric SIMD operations on these data types, these data types are interpreted as containing byte, word, or doubleword integer values.

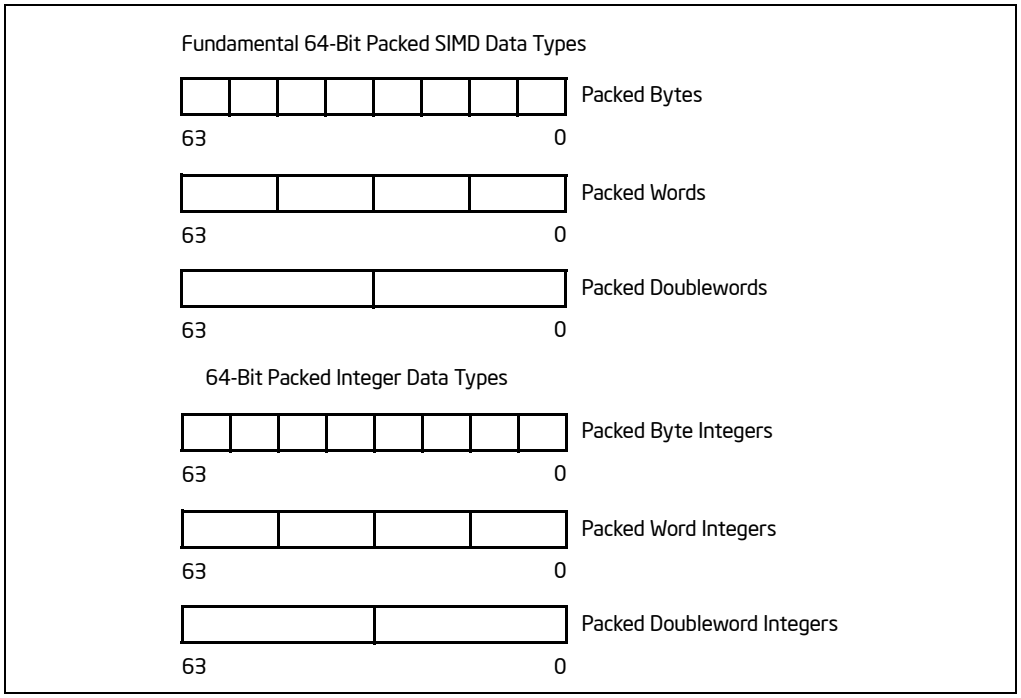


Figure 4-7. 64-Bit Packed SIMD Data Types

4.6.2 128-Bit Packed SIMD Data Types

The 128-bit packed SIMD data types were introduced into the IA-32 architecture in the SSE extensions and used with SSE2, SSE3 and SSSE3 extensions. They are operated on primarily in the 128-bit XMM registers and memory. The fundamental 128-bit packed data types are packed bytes, packed words, packed doublewords, and packed quadwords (see Figure 4-8). When performing SIMD operations on these fundamental data types in XMM registers, these data types are interpreted as containing packed or scalar single-precision floating-point or double-precision floating-point values, or as containing packed byte, word, doubleword, or quadword integer values.

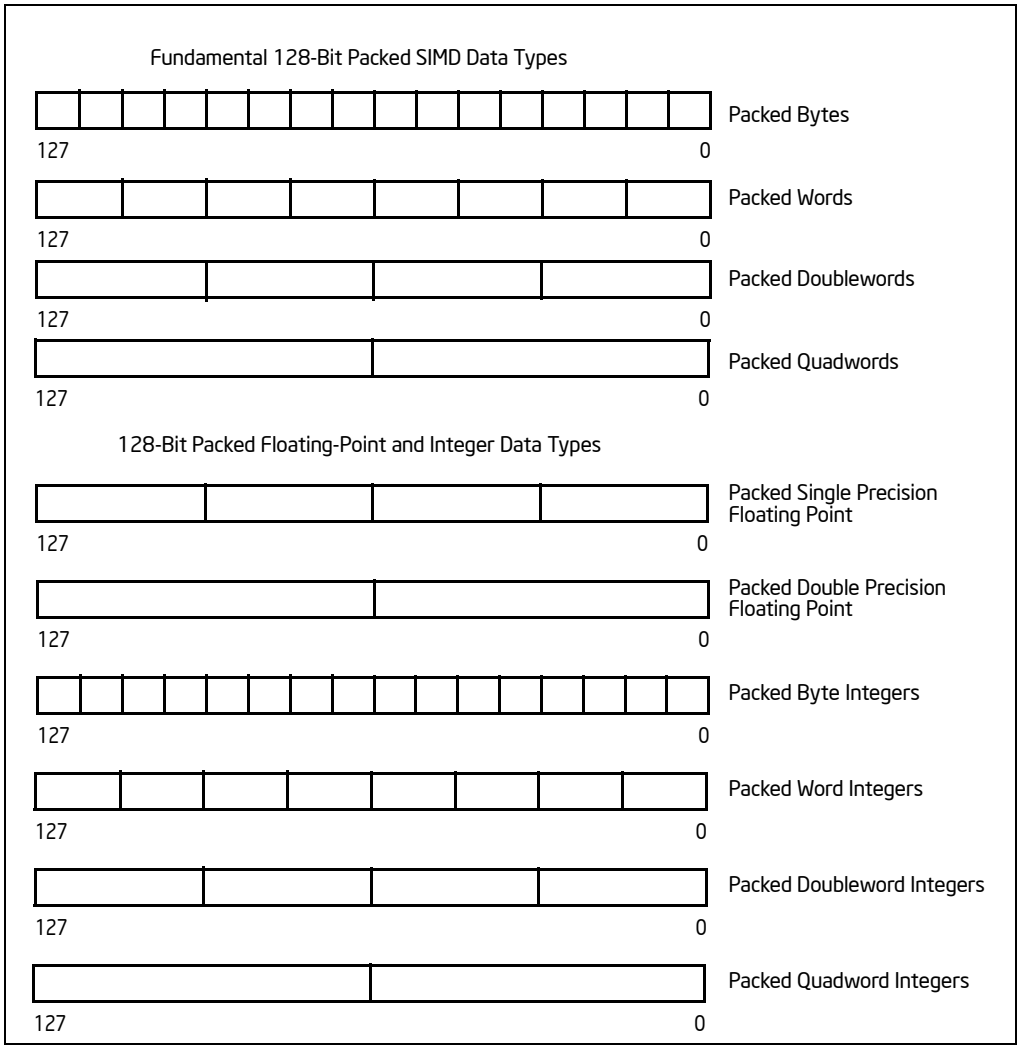


Figure 4-8. 128-Bit Packed SIMD Data Types

4.7 BCD AND PACKED BCD INTEGERS

Binary-coded decimal integers (BCD integers) are unsigned 4-bit integers with valid values ranging from 0 to 9. IA-32 architecture defines operations on BCD integers located in one or more general-purpose registers or in one or more x87 FPU registers (see Figure 4-9).

Figure 4-9. BCD Data Types

When operating on BCD integers in general-purpose registers, the BCD values can be unpacked (one BCD digit per byte) or packed (two BCD digits per byte). The value of an unpacked BCD integer is the binary value of the low half-byte (bits 0 through 3). The high half-byte (bits 4 through 7) can be any value during addition and subtraction, but must be zero during multiplication and division. Packed BCD integers allow two BCD digits to be contained in one byte. Here, the digit in the high half-byte is more significant than the digit in the low half-byte.

When operating on BCD integers in x87 FPU data registers, BCD values are packed in an 80-bit format and referred to as decimal integers. In this format, the first 9 bytes hold 18 BCD digits, 2 digits per byte. The least-significant digit is contained in the lower half-byte of byte 0 and the most-significant digit is contained in the upper half-byte of byte 9. The most significant bit of byte 10 contains the sign bit (0 = positive and 1 = negative; bits 0 through 6 of byte 10 are don't care bits). Negative decimal integers are not stored in two's complement form; they are distinguished from positive decimal integers only by the sign bit. The range of decimal integers that can be encoded in this format is $-10^{18}+1$ to $10^{18}-1$.

The decimal integer format exists in memory only. When a decimal integer is loaded in an x87 FPU data register, it is automatically converted to the double-extended-precision floating-point format. All decimal integers are exactly representable in double extended-precision format.

Table 4-4 gives the possible encodings of value in the decimal integer data type.

Table 4-4. Packed Decimal Integer Encodings

Class	Sign		Magnitude					
			digit	digit	digit	digit	...	digit
Positive								
Largest	0	0000000	1001	1001	1001	1001	...	1001
	.	.			.			
	.	.			.			
Smallest	0	0000000	0000	0000	0000	0000	...	0001
Zero	0	0000000	0000	0000	0000	0000	...	0000
Negative								
Zero	1	0000000	0000	0000	0000	0000	...	0000
Smallest	1	0000000	0000	0000	0000	0000	...	0001
	.	.			.			
	.	.			.			
Largest	1	0000000	1001	1001	1001	1001	...	1001

Table 4-4. Packed Decimal Integer Encodings (Contd.)

Packed BCD Integer Indefinite	1	1111111	1111	1111	1100	0000	...	0000
	← 1 byte →		← 9 bytes →					

The packed BCD integer indefinite encoding (FFFFC000000000000000H) is stored by the FBSTP instruction in response to a masked floating-point invalid-operation exception. Attempting to load this value with the FBLD instruction produces an undefined result.

4.8 REAL NUMBERS AND FLOATING-POINT FORMATS

This section describes how real numbers are represented in floating-point format in x87 FPU and SSE/SSE2/SSE3/SSE4.1 and Intel AVX floating-point instructions. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE Standard 754 for Binary Floating-Point Arithmetic may wish to skip this section.

4.8.1 Real Number System

As shown in Figure 4-10, the real-number system comprises the continuum of real numbers from minus infinity (– ∞) to plus infinity (+ ∞).

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number (floating-point) calculations. As shown at the bottom of Figure 4-10, the subset of real numbers that the IA-32 architecture supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the IEEE Standard 754 floating-point formats.

4.8.2 Floating-Point Format

To increase the speed and efficiency of real-number computations, computers and microprocessors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent (see Figure 4-11).

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The integer-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power by which the significand is multiplied.

Table 4-5 shows how the real number 178.125 (in ordinary decimal format) is stored in IEEE Standard 754 floating-point format. The table lists a progression of real number notations that leads to the single-precision, 32-bit floating-point format. In this format, the significand is normalized (see Section 4.8.2.1, "Normalized Numbers") and the exponent is biased (see Section 4.8.2.2, "Biased Exponent"). For the single-precision floating-point format, the biasing constant is +127.

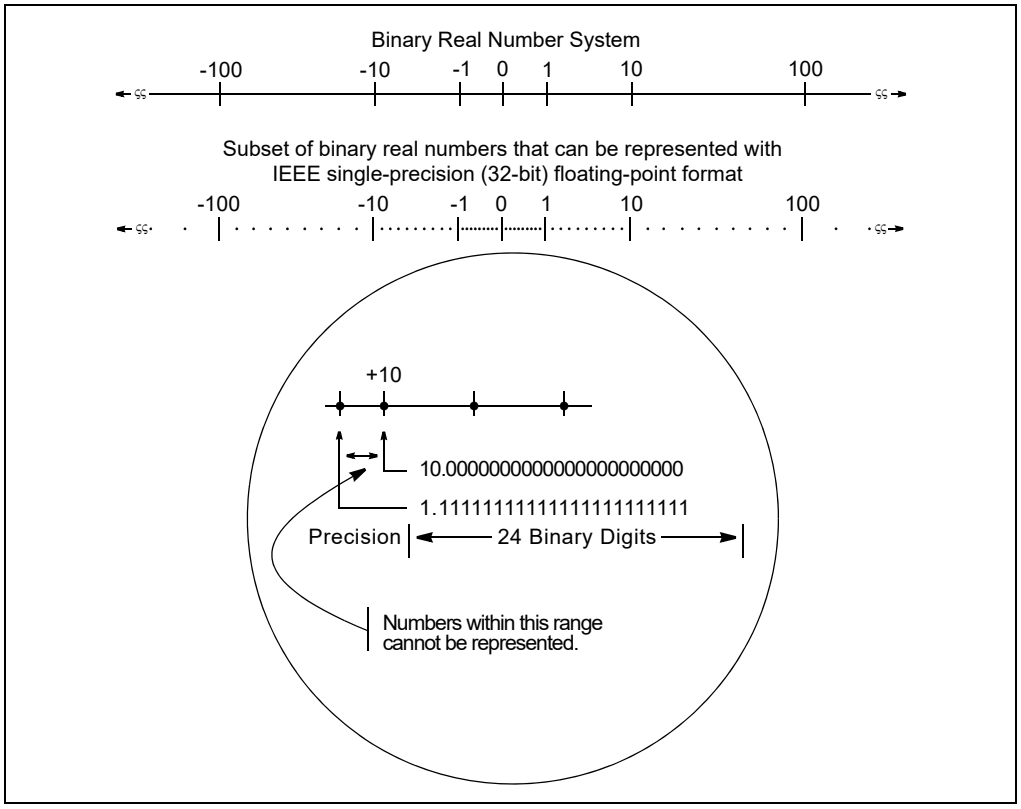


Figure 4-10. Binary Real Number System

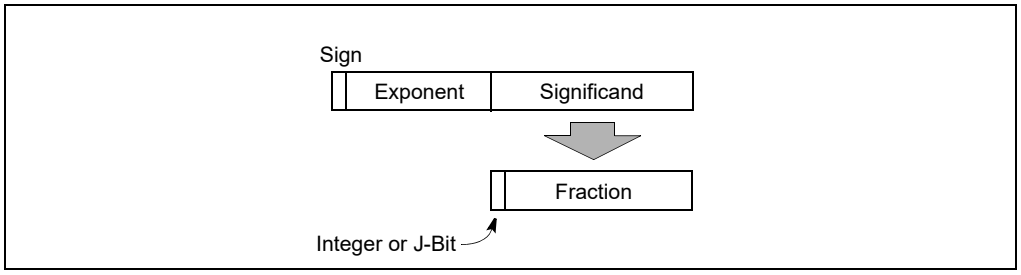


Figure 4-11. Binary Floating-Point Format

Table 4-5. Real and Floating-Point Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
IEEE Single-Precision Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 1. (Implied)

4.8.2.1 Normalized Numbers

In most cases, floating-point numbers are encoded in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

4.8.2.2 Biased Exponent

In the IA-32 architecture, the exponents of floating-point numbers are encoded in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

See Section 4.2.2, "Floating-Point Data Types," for a list of the biasing constants that the IA-32 architecture uses for the various sizes of floating-point data-types.

4.8.3 Real Number and Non-number Encodings

A variety of real numbers and special values can be encoded in the IEEE Standard 754 floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros
- Denormalized finite numbers
- Normalized finite numbers
- Signed infinities
- NaNs
- Indefinite numbers

(The term NaN stands for "Not a Number.")

Figure 4-12 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision floating-point format. The term "S" indicates the sign bit, "E" the biased exponent, and "Sig" the significand. The exponent values are given in decimal. The integer bit is shown for the significands, even though the integer bit is implied in single-precision floating-point format.

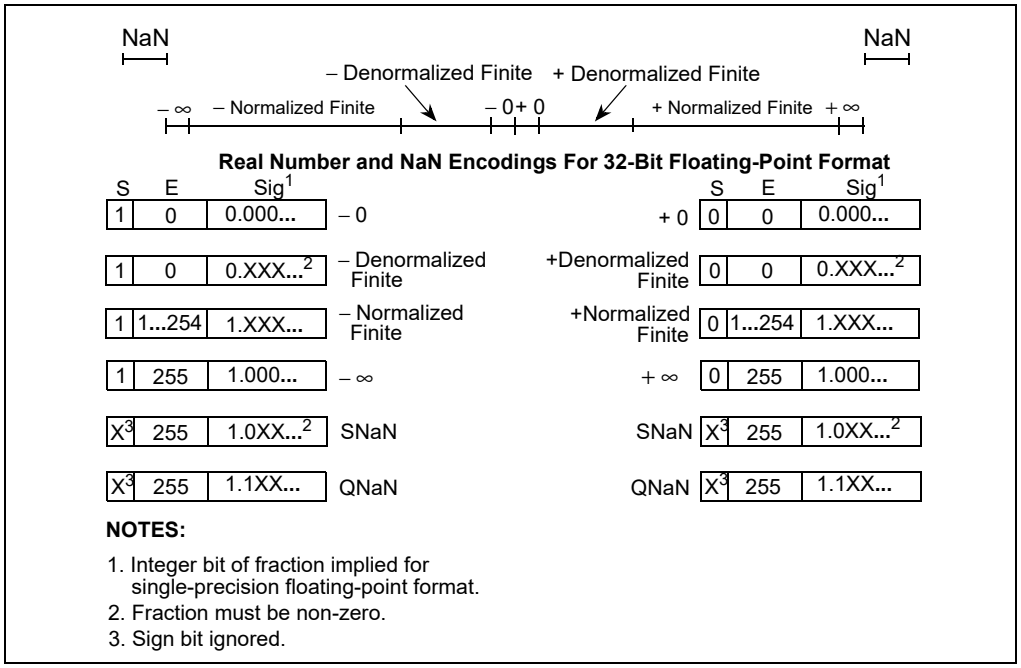


Figure 4-12. Real Numbers and NaNs

An IA-32 processor can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

4.8.3.1 Signed Zeros

Zero can be represented as a +0 or a –0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

4.8.3.2 Normalized and Denormalized Finite Numbers

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the single-precision floating-point format shown in Figure 4-12, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

When floating-point numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called **denormalized** numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization may cause a loss of precision (the number of significant bits is reduced by the leading zeros).

When performing normalized floating-point computations, an IA-32 processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an **underflow** condition. The exact conditions are specified in Section 4.9.1.5, "Numeric Underflow Exception (#U)."

A denormalized number is computed through a technique called gradual underflow. Table 4-6 gives an example of gradual underflow in the denormalization process. Here the single-precision format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a

normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 4-6. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

* Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The Intel 64 and IA-32 architectures deal with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

4.8.3.3 Signed Infinities

The two infinities, + ∞ and – ∞, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a significand of 1.00...00 (the integer bit may be implied) and the maximum biased exponent allowed in the specified format (for example, 255_{10} for the single-precision format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, –∞ is less than any finite number and +∞ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers may represent an underflow condition, the two ∞ numbers may represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

4.8.3.4 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 4-12, the encoding space for NaNs in the floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction (the sign bit is ignored for NaNs).

The IA-32 architecture defines two classes of NaNs: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal a floating-point invalid-operation exception whenever they appear as operands in arithmetic operations.

SNaNs are typically used to trap or invoke an exception handler. They must be inserted by software; that is, the processor never generates an SNaN as a result of a floating-point operation.

4.8.3.5 Operating on SNaNs and QNaNs

When a floating-point operation is performed on an SNaN and/or a QNaN, the result of the operation is either a QNaN delivered to the destination operand or the generation of a floating-point invalid operation exception, depending on the following rules:

- If one of the source operands is an SNaN and the floating-point invalid-operation exception is not masked (see Section 4.9.1.1, "Invalid Operation Exception (#I)"), then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand. If one of the source operands is a QNaN and the floating-point invalid-operation exception is not masked and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)", then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand.
- If either or both of the source operands are NaNs and floating-point invalid-operation exception is masked, the result is as shown in Table 4-7. When an SNaN is converted to a QNaN, the conversion is handled by setting the most-significant fraction bit of the SNaN to 1. Also, when one of the source operands is an SNaN, or when it is is a QNaN and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)", then the floating-point invalid-operation exception flag is set. Note that for some combinations of source operands, the result is different for x87 FPU operations and for SSE/SSE2/SSE3/SSE4.1 operations. Intel AVX follows the same behavior as SSE/SSE2/SSE3/SSE4.1 in this respect.
- When neither of the source operands is a NaN, but the operation generates a floating-point invalid-operation exception (see Tables 8-10 and 11-1), the result is commonly a QNaN FP Indefinite (Section 4.8.3.7).

Any exceptions to the behavior described in Table 4-7 are described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," and Section 11.5.2.1, "Invalid Operation Exception (#I)."

Table 4-7. Rules for Handling NaNs

Source Operands	Result[1]
SNaN and QNaN	x87 FPU — QNaN source operand. SSE/SSE2/SSE3/SSE4.1/AVX — First source operand (if this operand is an SNaN, it is converted to a QNaN)
Two SNaNs	x87 FPU—SNaN source operand with the larger significand, converted into a QNaN SSE/SSE2/SSE3/SSE4.1/AVX — First source operand converted to a QNaN
Two QNaNs	x87 FPU — QNaN source operand with the larger significand SSE/SSE2/SSE3/SSE4.1/AVX — First source operand
SNaN and a floating-point value	SNaN source operand, converted into a QNaN
QNaN and a floating-point value	QNaN source operand
SNaN (for instructions that take only one operand)	SNaN source operand, converted into a QNaN
QNaN (for instructions that take only one operand)	QNaN source operand

NOTE:

1. For SSE/SSE2/SSE3/SSE4.1 instructions, the first operand is generally a source operand that becomes the destination operand. For AVX instructions, the first source operand is usually the 2nd operand in a non-destructive source syntax. Within the **Result** column, the x87 FPU notation also applies to the FISTTP instruction in SSE3; the SSE3 notation applies to the SIMD floating-point instructions.

4.8.3.6 Using SNaNs and QNaNs in Applications

Except for the rules given at the beginning of Section 4.8.3.4, "NaNs," for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.

By unmasking the invalid operation exception, the programmer can use signaling NaNs to trap to the exception handler. The generality of this approach and the large number of NaN values that are available provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler can use signaling NaNs as references to uninitialized (real) array elements. The compiler can preinitialize each array element with a signaling NaN whose significand contains the index (relative position) of the element. Then, if an application program attempts to access an element that it has not initialized, it can use the NaN placed there by the compiler. If the invalid operation exception is unmasked, an interrupt will occur, and the exception handler will be invoked. The exception handler can determine which element has been accessed, since the operand address field of the exception pointer will point to the NaN, and the NaN will contain the index number of the array element.

Quiet NaNs are often used to speed up debugging. In its early testing phase, a program often contains multiple errors. An exception handler can be written to save diagnostic information in memory whenever it is invoked. After storing the diagnostic data, it can supply a quiet NaN as the result of the erroneous instruction, and that NaN can point to its associated diagnostic area in memory. The program will then continue, creating a different NaN for each error. When the program ends, the NaN results can be used to access the diagnostic data saved at the time the errors occurred. Many errors can thus be diagnosed and corrected in one test run.

In embedded applications that use computed results in further computations, an undetected QNaN can invalidate all subsequent results. Such applications should therefore periodically check for QNaNs and provide a recovery mechanism to be used if a QNaN result is detected.

4.8.3.7 QNaN Floating-Point Indefinite

For the floating-point data type encodings (single-precision, double-precision, and double-extended-precision), one unique encoding (a QNaN) is reserved for representing the special value QNaN floating-point indefinite. The x87 FPU and the SSE/SSE2/SSE3/SSE4.1/AVX extensions return these indefinite values as responses to some masked floating-point exceptions. Table 4-3 shows the encoding used for the QNaN floating-point indefinite.

4.8.3.8 Half-Precision Floating-Point Operation

Half-precision floating-point values are not used by the processor directly for arithmetic operations. Two instructions, VCVTPH2PS, VCVTPS2PH, provide conversion only between half-precision and single-precision floating-point values.

The SIMD floating-point exception behavior of VCVTPH2PS and VCVTPS2PH are described in Section 14.4.1.

4.8.4 Rounding

When performing floating-point operations, the processor produces an infinitely precise floating-point result in the destination format (single-precision, double-precision, or double extended-precision floating-point) whenever possible. However, because only a subset of the numbers in the real number continuum can be represented in IEEE Standard 754 floating-point formats, it is often the case that an infinitely precise result cannot be encoded exactly in the format of the destination operand.

For example, the following value (*a*) has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the single-precision format (which has only a 23-bit fraction):

(*a*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$

To round this result (*a*), the processor first selects two representable fractions *b* and *c* that most closely bracket *a* in value ($b < a < c$).

(*b*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$

(*c*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$

The processor then sets the result to *b* or to *c* according to the selected rounding mode. Rounding introduces an error in a result that is less than one unit in the last place (the least significant bit position of the floating-point value) to which the result is rounded.

The IEEE Standard 754 defines four rounding modes (see Table 4-8): round to nearest, round up, round down, and round toward zero. The default rounding mode (for the Intel 64 and IA-32 architectures) is round to nearest. This mode provides the most accurate and statistically unbiased estimate of the true result and is suitable for most applications.

Table 4-8. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default
Round down (toward $-\infty$)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward $+\infty$)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

The round up and round down modes are termed **directed rounding** and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multistep computation, when the intermediate results of the computation are subject to rounding.

The round toward zero mode (sometimes called the "chop" mode) is commonly used when performing integer arithmetic with the x87 FPU.

The rounded result is called the inexact result. When the processor produces an inexact result, the floating-point precision (inexact) flag (PE) is set (see Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)").

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

4.8.4.1 Rounding Control (RC) Fields

In the Intel 64 and IA-32 architectures, the rounding mode is controlled by a 2-bit rounding-control (RC) field (Table 4-8 shows the encoding of this field). The RC field is implemented in two different locations:

- x87 FPU control register (bits 10 and 11)
- The MXCSR register (bits 13 and 14)

Although these two RC fields perform the same function, they control rounding for different execution environments within the processor. The RC field in the x87 FPU control register controls rounding for computations performed with the x87 FPU instructions; the RC field in the MXCSR register controls rounding for SIMD floating-point computations performed with the SSE/SSE2 instructions.

4.8.4.2 Truncation with SSE and SSE2 Conversion Instructions

The following SSE/SSE2 instructions automatically truncate the results of conversions from floating-point values to integers when the result it inexact: CVTTPD2DQ, CVTTPS2DQ, CVTTPD2PI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI. Here, truncation means the round toward zero mode described in Table 4-8.

4.9 OVERVIEW OF FLOATING-POINT EXCEPTIONS

The following section provides an overview of floating-point exceptions and their handling in the IA-32 architecture. For information specific to the x87 FPU and to the SSE/SSE2/SSE3/SSE4.1 extensions, refer to the following sections:

- Section 8.4, "x87 FPU Floating-Point Exception Handling"

- Section 11.5, "SSE, SSE2, and SSE3 Exceptions"

When operating on floating-point operands, the IA-32 architecture recognizes and detects six classes of exception conditions:

- Invalid operation (#I)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

The nomenclature of "#" symbol followed by one or two letters (for example, #P) is used in this manual to indicate exception conditions. It is merely a short-hand form and is not related to assembler mnemonics.

NOTE

All of the exceptions listed above except the denormal-operand exception (#D) are defined in IEEE Standard 754.

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 precision exceptions are post-computation exceptions.

Each of the six exception classes has a corresponding flag bit (IE, ZE, OE, UE, DE, or PE) and mask bit (IM, ZM, OM, UM, DM, or PM). When one or more floating-point exception conditions are detected, the processor sets the appropriate flag bits, then takes one of two possible courses of action, depending on the settings of the corresponding mask bits:

- Mask bit set. Handles the exception automatically, producing a predefined (and often times usable) result, while allowing program execution to continue undisturbed.
- Mask bit clear. Invokes a software exception handler to handle the exception.

The masked (default) responses to exceptions have been chosen to deliver a reasonable result for each exception condition and are generally satisfactory for most floating-point applications. By masking or unmasking specific floating-point exceptions, programmers can delegate responsibility for most exceptions to the processor and reserve the most severe exception conditions for software exception handlers.

Because the exception flags are "sticky," they provide a cumulative record of the exceptions that have occurred since they were last cleared. A programmer can thus mask all exceptions, run a calculation, and then inspect the exception flags to see if any exceptions were detected during the calculation.

In the IA-32 architecture, floating-point exception flag and mask bits are implemented in two different locations:

- x87 FPU status word and control word. The flag bits are located at bits 0 through 5 of the x87 FPU status word and the mask bits are located at bits 0 through 5 of the x87 FPU control word (see Figures 8-4 and 8-6).
- MXCSR register. The flag bits are located at bits 0 through 5 of the MXCSR register and the mask bits are located at bits 7 through 12 of the register (see Figure 10-3).

Although these two sets of flag and mask bits perform the same function, they report on and control exceptions for different execution environments within the processor. The flag and mask bits in the x87 FPU status and control words control exception reporting and masking for computations performed with the x87 FPU instructions; the companion bits in the MXCSR register control exception reporting and masking for SIMD floating-point computations performed with the SSE/SSE2/SSE3 instructions.

Note that when exceptions are masked, the processor may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the processor can detect a denormalized operand, perform its masked response to this exception, and then detect numeric underflow.

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

4.9.1 Floating-Point Exception Conditions

The following sections describe the various conditions that cause a floating-point exception to be generated and the masked response of the processor when these conditions are detected. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A & 3B,* list the floating-point exceptions that can be signaled for each floating-point instruction.

4.9.1.1 Invalid Operation Exception (#I)

The processor reports an invalid operation exception in response to one or more invalid arithmetic operands. If the invalid operation exception is masked, the processor sets the IE flag and returns an indefinite value or a QNaN. This value overwrites the destination register specified by the instruction. If the invalid operation exception is not masked, the IE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See Section 4.8.3.6, "Using SNaNs and QNaNs in Applications," for information about the result returned when an exception is caused by an SNaN.

The processor can detect a variety of invalid arithmetic operations that can be coded in a program. These operations generally indicate a programming error, such as dividing ∞ by ∞. See the following sections for information regarding the invalid-operation exception when detected while executing x87 FPU or SSE/SSE2/SSE3/SSE4.1 or AVX instructions:

- x87 FPU; Section 8.5.1, "Invalid Operation Exception".
- SIMD floating-point exceptions; Section 11.5.2.1, "Invalid Operation Exception (#I)".

4.9.1.2 Denormal Operand Exception (#D)

The processor reports the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers"). When the exception is masked, the processor sets the DE flag and proceeds with the instruction. Operating on denormal numbers will produce results at least as good as, and often better than, what can be obtained when denormal numbers are flushed to zero. Programmers can mask this exception so that a computation may proceed, then analyze any loss of accuracy when the final result is delivered.

When a denormal-operand exception is not masked, the DE flag is set, a software exception handler is invoked, and the operands remain unaltered. When denormal operands have reduced significance due to loss of low-order bits, it may be advisable to not operate on them. Precluding denormal operands from computations can be accomplished by an exception handler that responds to unmasked denormal-operand exceptions.

See the following sections for information regarding the denormal-operand exception when detected while executing x87 FPU or SSE/SSE2/SSE3 instructions:

- x87 FPU; Section 8.5.2, "Denormal Operand Exception (#D)".
- SIMD floating-point exceptions; Section 11.5.2.2, "Denormal-Operand Exception (#D)".

4.9.1.3 Divide-By-Zero Exception (#Z)

The processor reports the floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The masked response for the divide-by-zero exception is to set the ZE flag and return an infinity signed with the exclusive OR of the sign of the operands. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See the following sections for information regarding the divide-by-zero exception when detected while executing x87 FPU or SSE/SSE2 instructions:

- x87 FPU; Section 8.5.3, "Divide-By-Zero Exception (#Z)".
- SIMD floating-point exceptions; Section 11.5.2.3, "Divide-By-Zero Exception (#Z)".

4.9.1.4 Numeric Overflow Exception (#O)

The processor reports a floating-point numeric overflow exception whenever the rounded result of an instruction exceeds the largest allowable finite value that will fit into the destination operand. Table 4-9 shows the threshold range for numeric overflow for each of the floating-point formats; overflow occurs when a rounded result falls at or outside this threshold range.

Table 4-9. Numeric Overflow Thresholds

Floating-Point Format	Overflow Thresholds
Single Precision	$\|x\| \geq 1.0 * 2^{128}$
Double Precision	$\|x\| \geq 1.0 * 2^{1024}$
Double Extended Precision	$\|x\| \geq 1.0 * 2^{16384}$

When a numeric-overflow exception occurs and the exception is masked, the processor sets the OE flag and returns one of the values shown in Table 4-10, according to the current rounding mode. See Section 4.8.4, "Rounding."

When numeric overflow occurs and the numeric-overflow exception is not masked, the OE flag is set, a software exception handler is invoked, and the source and destination operands either remain unchanged or a biased result is stored in the destination operand (depending whether the overflow exception was generated during an SSE/SSE2/SSE3 floating-point operation or an x87 FPU operation).

Table 4-10. Masked Responses to Numeric Overflow

Rounding Mode	Sign of True Result	Result
To nearest	+	+∞
	–	–∞
Toward –∞	+	Largest finite positive number
	–	–∞
Toward +∞	+	+∞
	–	Largest finite negative number
Toward zero	+	Largest finite positive number
	–	Largest finite negative number

See the following sections for information regarding the numeric overflow exception when detected while executing x87 FPU instructions or while executing SSE/SSE2/SSE3 instructions:

- x87 FPU; Section 8.5.4, "Numeric Overflow Exception (#O)"
- SIMD floating-point exceptions; Section 11.5.2.4, "Numeric Overflow Exception (#O)"

4.9.1.5 Numeric Underflow Exception (#U)

The processor detects a potential floating-point numeric underflow condition whenever the result of rounding with unbounded exponent (taking into account precision control for x87) is non-zero and tiny; that is, non-zero and less than the smallest possible normalized, finite value that will fit into the destination operand. Table 4-11 shows the threshold range for numeric underflow for each of the floating-point formats (assuming normalized results); underflow occurs when a rounded result falls strictly within the threshold range. The ability to detect and handle underflow is provided to prevent a very small result from propagating through a computation and causing another exception (such as overflow during division) to be generated at a later time. Results which trigger underflow are also potentially less accurate.

Table 4-11. Numeric Underflow (Normalized) Thresholds

Floating-Point Format	Underflow Thresholds*
Single Precision	$\|x\| < 1.0 * 2^{-126}$
Double Precision	$\|x\| < 1.0 * 2^{-1022}$
Double Extended Precision	$\|x\| < 1.0 * 2^{-16382}$

* Where 'x' is the result rounded to destination precision with an unbounded exponent range.

How the processor handles an underflow condition, depends on two related conditions:

- creation of a tiny, non-zero result
- creation of an inexact result; that is, a result that cannot be represented exactly in the destination format

Which of these events causes an underflow exception to be reported and how the processor responds to the exception condition depends on whether the underflow exception is masked:

- **Underflow exception masked —** The underflow exception is reported (the UE flag is set) only when the result is both tiny and inexact. The processor returns a correctly signed result whose magnitude is less than or equal to the smallest positive normal floating-point number to the destination operand, regardless of inexactness.
- **Underflow exception not masked —** The underflow exception is reported when the result is non-zero tiny, regardless of inexactness. The processor leaves the source and destination operands unaltered or stores a biased result in the destination operand (depending whether the underflow exception was generated during an SSE/SSE2/SSE3 floating-point operation or an x87 FPU operation) and invokes a software exception handler.

See the following sections for information regarding the numeric underflow exception when detected while executing x87 FPU instructions or while executing SSE/SSE2/SSE3 instructions:

- x87 FPU; Section 8.5.5, "Numeric Underflow Exception (#U)"
- SIMD floating-point exceptions; Section 11.5.2.5, "Numeric Underflow Exception (#U)"

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

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary floating-point form. This exception occurs frequently and indicates that some (normally acceptable) accuracy will be lost due to rounding. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the processor sets the PE flag and stores the rounded result in the destination operand. The current rounding mode determines the method used to round the result. See Section 4.8.4, "Rounding."

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the PE flag is set, the rounded result is stored in the destination operand, and a software exception handler is invoked.

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

- If an inexact result occurs along with masked overflow or underflow, the OE flag or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions; see Section 4.9.1.4, "Numeric Overflow Exception (#O)," or Section 4.9.1.5, "Numeric Underflow Exception (#U)." If the inexact result exception is unmasked, the processor also invokes a software exception handler.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions, and a software exception handler is invoked.

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

See the following sections for information regarding the inexact-result exception when detected while executing x87 FPU or SSE/SSE2/SSE3 instructions:

- x87 FPU; Section 8.5.6, "Inexact-Result (Precision) Exception (#P)"
- SIMD floating-point exceptions; Section 11.5.2.3, "Divide-By-Zero Exception (#Z)"

4.9.2 Floating-Point Exception Priority

The processor handles exceptions according to a predetermined precedence. When an instruction generates two or more exception conditions, the exception precedence sometimes results in the higher-priority exception being handled and the lower-priority exceptions being ignored. For example, dividing an SNaN by zero can potentially signal an invalid-operation exception (due to the SNaN operand) and a divide-by-zero exception. Here, if both exceptions are masked, the processor handles the higher-priority exception only (the invalid-operation exception), returning a QNaN to the destination. Alternately, a denormal-operand or inexact-result exception can accompany a numeric underflow or overflow exception with both exceptions being handled.

The precedence for floating-point exceptions is as follows:

1. Invalid-operation exception, subdivided as follows:
   a. stack underflow (occurs with x87 FPU only)
   b. stack overflow (occurs with x87 FPU only)
   c. operand of unsupported format (occurs with x87 FPU only when using the double extended-precision floating-point format)
   d. SNaN operand
2. QNaN operand. Though 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 zero-divide exception.
3. Any other invalid-operation exception not mentioned above or a divide-by-zero exception.
4. Denormal-operand exception. If masked, then instruction execution continues and a lower-priority exception can occur as well.
5. Numeric overflow and underflow exceptions; possibly in conjunction with the inexact-result exception.
6. Inexact-result exception.

Invalid operation, zero divide, and denormal operand exceptions are detected before a floating-point operation begins. Overflow, underflow, and precision exceptions are not detected until a true result has been computed. When an unmasked **pre-operation** exception is detected, the destination operand has not yet been updated, and appears as if the offending instruction has not been executed. When an unmasked **post-operation** exception is detected, the destination operand may be updated with a result, depending on the nature of the exception (except for SSE/SSE2/SSE3 instructions, which do not update their destination operands in such cases).

4.9.3 Typical Actions of a Floating-Point Exception Handler

After the floating-point exception handler is invoked, the processor handles the exception in the same manner that it handles non-floating-point exceptions. The floating-point exception handler is normally part of the operating system or executive software, and it usually invokes a user-registered floating-point exception handle.

A typical action of the exception handler is to store state information in memory. Other typical exception handler actions include:

- Examining the stored state information to determine the nature of the error
- Taking actions to correct the condition that caused the error
- Clearing the exception flags
- Returning to the interrupted program and resuming normal execution

In lieu of writing recovery procedures, the exception handler can do the following:

- Increment in software an exception counter for later display or printing
- Print or display diagnostic information (such as the state information)
- Halt further program execution

3. Updates to Chapter 5, Volume 1

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

Change to this chapter: Minor typo correction to introduction and added Intel SHA Extension row to Table 5-2 "Recent 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:

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

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.
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.

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

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

Table 5-2. Recent Instruction Set Extensions 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, T9000 series.
SSE4.2 Extensions, CRC32, POPCNT	Intel Core i7 965 processor, Intel Xeon processors X3400, X3500, X5500, X6500, X7500 series.
AESNI, PCLMULQDQ	Intel Xeon processor E7 series, Intel Xeon processors X3600, X5600, Intel Core i7 980X processor; Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel Core processor families.
Intel AVX	Intel Xeon processor E3 and E5 families; 2nd Generation Intel Core i7, i5, i3 processor 2xxx families.
F16C, RDRAND, FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Next Generation Intel Xeon processors, Intel Xeon processor E5 v2 and E7 v2 families.
FMA, AVX2, BMI1, BMI2, INVPCID	Intel Xeon processor E3-1200 v3 product family; 4th Generation Intel Core processor family.
TSX	Intel Xeon processor E7 v3 product family.
Intel SHA Extensions	Intel Atom processor based on Goldmont microarchitecture.
ADX, RDSEED, CLAC, STAC	Intel Core M processor family; 5th Generation Intel Core processor family.
CLFLUSHOPT, XSAVEC, XSAVES, MPX, SGX1	6th Generation Intel Core processor family.

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 preform 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 following 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.
CMOVB/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.

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

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.

5.1.8 String Instructions

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

MOVS/MOVSB	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/POPFD	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 Sate 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.

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.

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

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.

1. SSE3 provides an instruction FISTTP for integer conversion.

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

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.
PADDD	Add packed doubleword integers.
PADDSB	Add packed signed byte integers with signed saturation.
PADDSW	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.
PSUBSW	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.
CVTPS2PI	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTPS2PI	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.
PMOVMSKB	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 PREFETCH*h* 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.
PREFETCH*h*	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 preform 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.

SHUFPD	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.
CVTPS2PD	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.
PSHUFD	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

ADDSUBPS	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.
ADDSUBPD	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 — Permutes 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.
PMOVZXBD	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 AESNI AND PCLMULQDQ

Six AESNI 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 VCVTPS2PH, VCVTPH2PS:

VCVTPH2PS	Convert eight/four data element containing 16-bit floating-point data into eight/four single-precision floating-point data.
VCVTPS2PH	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.
SHA1RNDS4	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.
SHA256RNDS2	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 enhance-

ment 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.
VEXPANDPD/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.
VPBROADCASTD/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.
VPTESTNMD/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.
VPTESTMD/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)	Lock Bus.
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMC	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
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[SENTER]	Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[SEXIT]	Exits the MLE.
GETSEC[PARAMETERS]	Returns SMX related parameter information.
GETSEC[SMCRTL]	SMX mode control.
GETSEC[WAKEUP]	Wakes up sleeping logical processors inside an MLE.

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.
BNDCU	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[EDBGRD]	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		

4. Updates to Chapter 8, Volume 1

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

Change to this chapter: Defined the "NPX" acronym: Numeric Processor Extensions.

CHAPTER 8
PROGRAMMING WITH THE X87 FPU

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

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

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

8.1 X87 FPU EXECUTION ENVIRONMENT

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

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

These registers are described in the following sections.

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

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

8.1.1 x87 FPU in 64-Bit Mode and Compatibility Mode

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

8.1.2 x87 FPU Data Registers

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

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

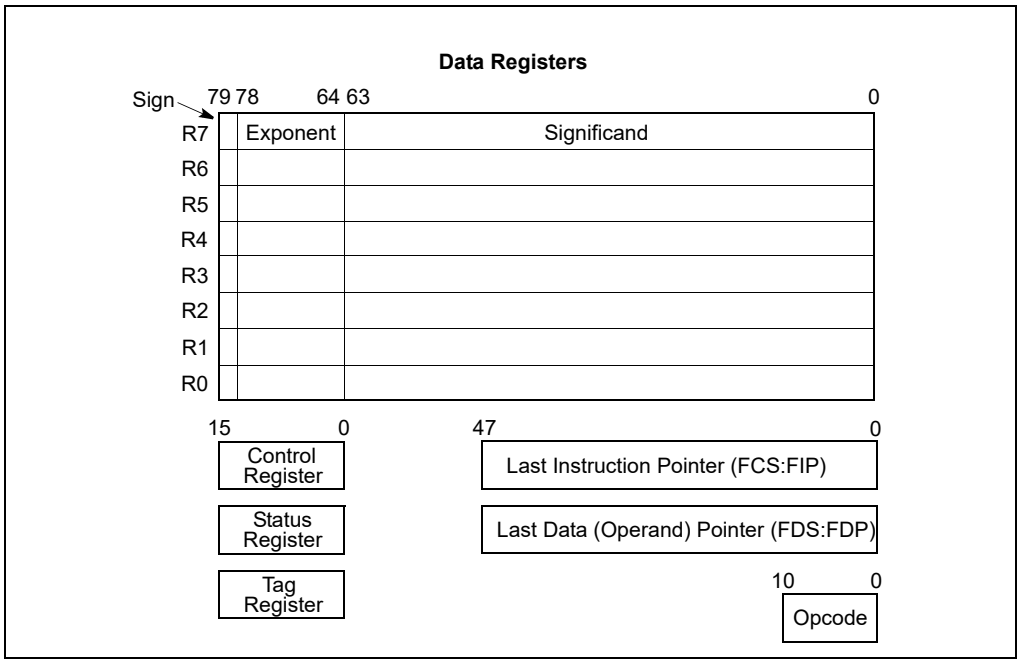


Figure 8-1. x87 FPU Execution Environment

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

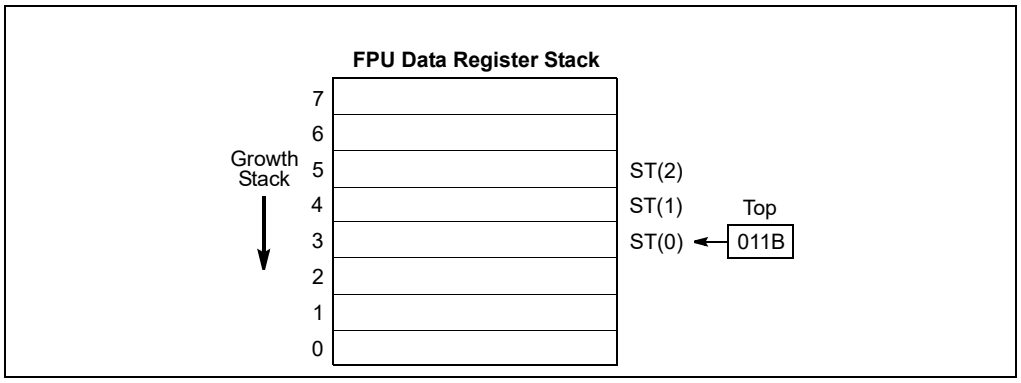


Figure 8-2. x87 FPU Data Register Stack

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

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

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

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

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

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

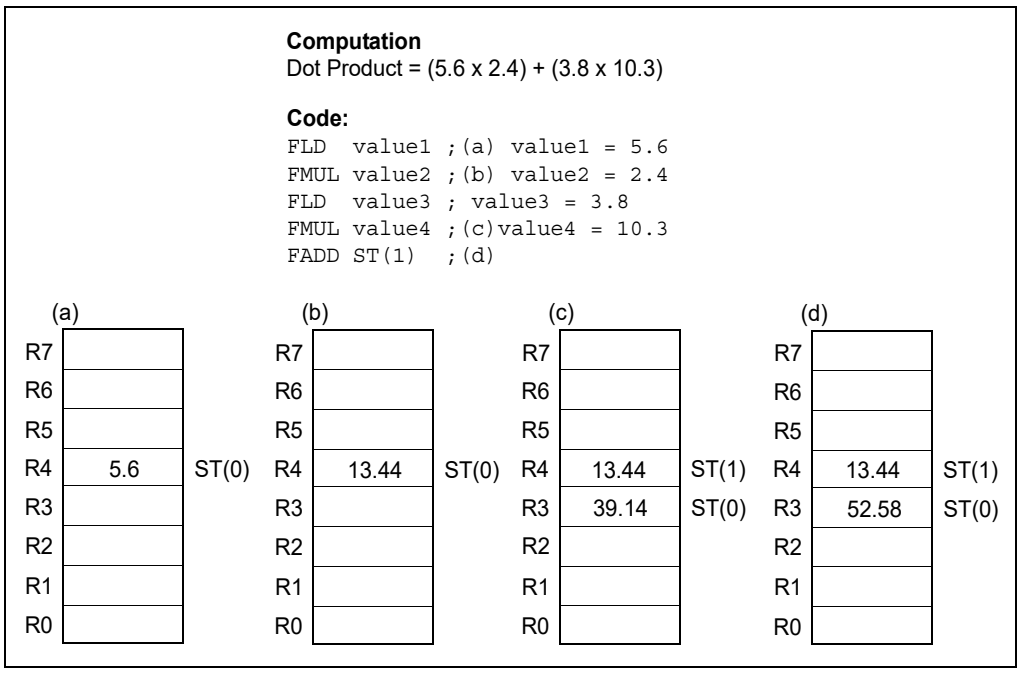


Figure 8-3. Example x87 FPU Dot Product Computation

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

8.1.2.1 Parameter Passing With the x87 FPU Register Stack

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

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

8.1.3 x87 FPU Status Register

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

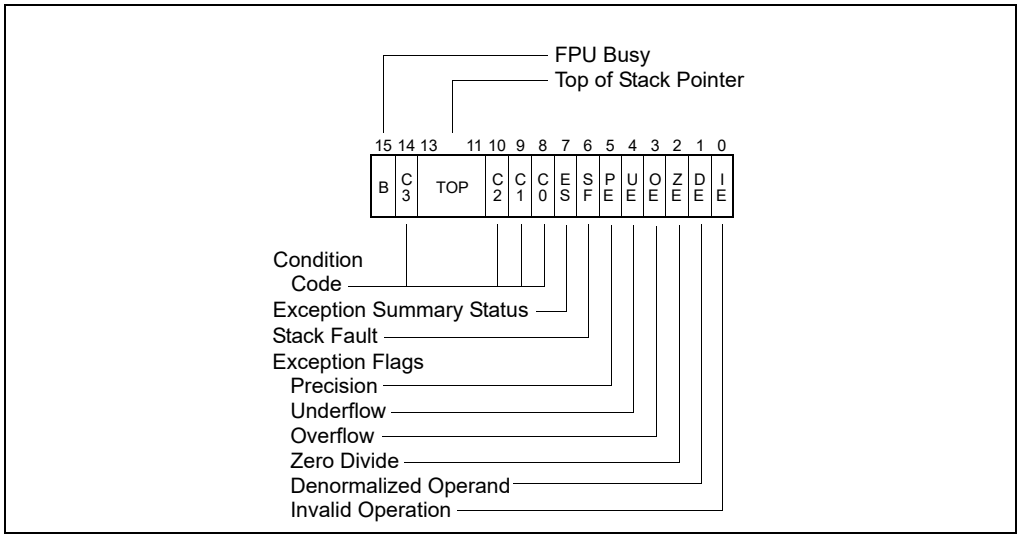


Figure 8-4. x87 FPU Status Word

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

8.1.3.1 Top of Stack (TOP) Pointer

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

8.1.3.2 Condition Code Flags

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

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

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

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

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

8.1.3.3 x87 FPU Floating-Point Exception Flags

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

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

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

Table 8-1. Condition Code Interpretation

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

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

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

8.1.3.4 Stack Fault Flag

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

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

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

8.1.4 Branching and Conditional Moves on Condition Codes

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

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

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

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

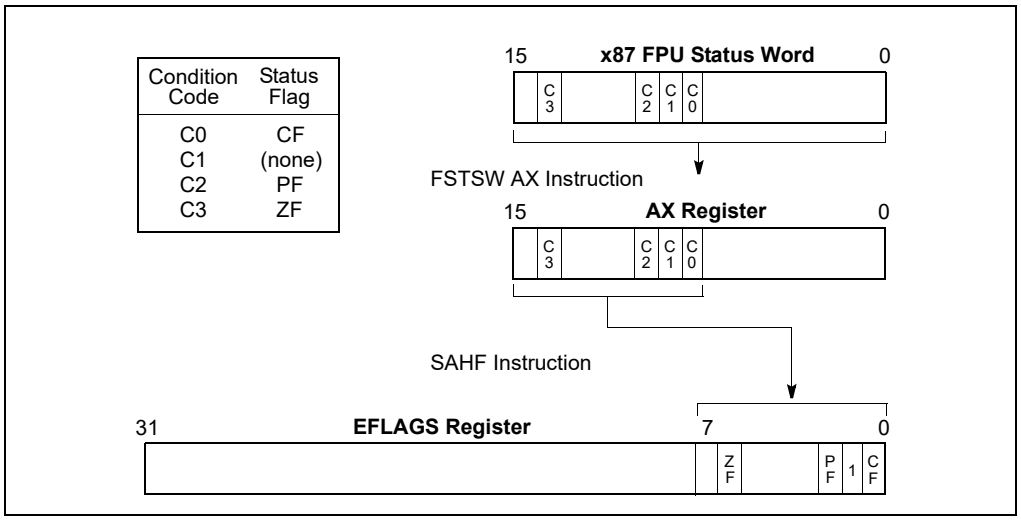


Figure 8-5. Moving the Condition Codes to the EFLAGS Register

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

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

8.1.5 x87 FPU Control Word

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

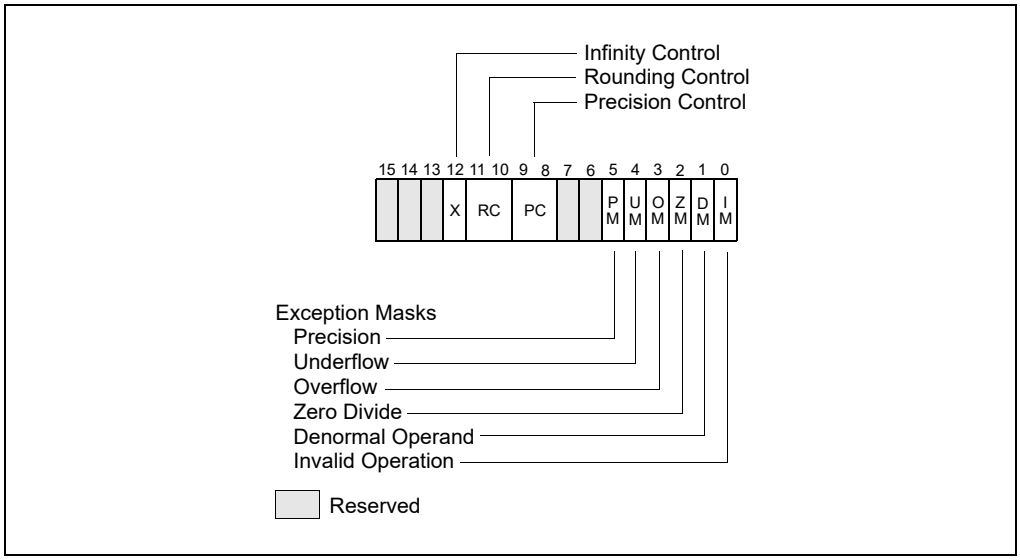


Figure 8-6. x87 FPU Control Word

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

8.1.5.1 x87 FPU Floating-Point Exception Mask Bits

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

8.1.5.2 Precision Control Field

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

Table 8-2. Precision Control Field (PC)

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

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

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

8.1.5.3 Rounding Control Field

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

8.1.6 Infinity Control Flag

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

8.1.7 x87 FPU Tag Word

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

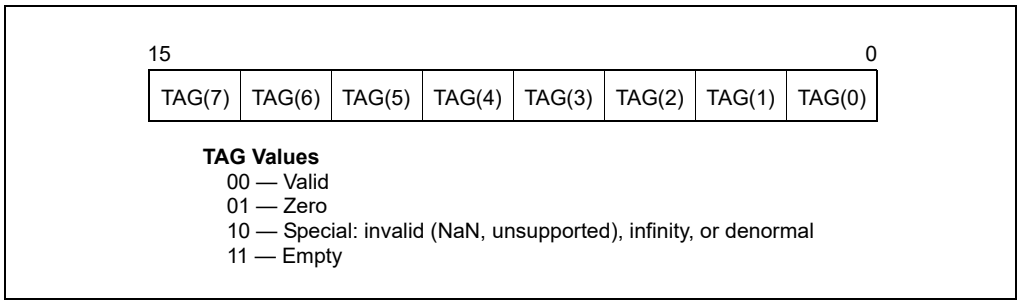


Figure 8-7. x87 FPU Tag Word

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

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

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

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

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

8.1.8 x87 FPU Instruction and Data (Operand) Pointers

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

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

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

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

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

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

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

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

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

8.1.9 Last Instruction Opcode

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

8.1.9.1 Fopcode Compatibility Sub-mode

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

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

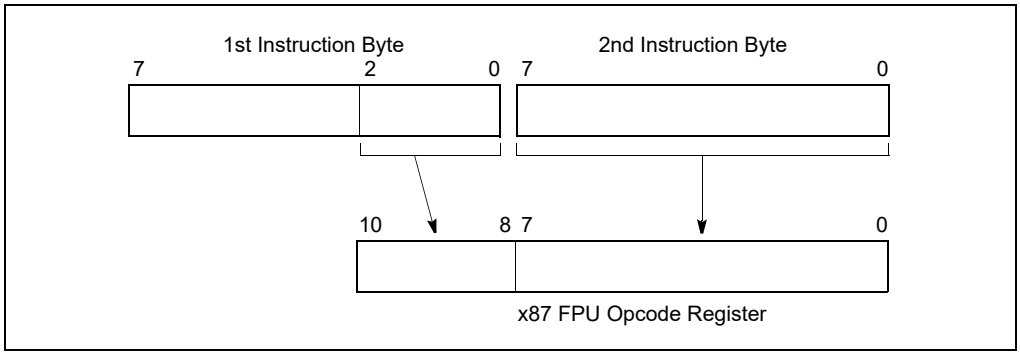


Figure 8-8. Contents of x87 FPU Opcode Registers

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

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

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

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

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

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

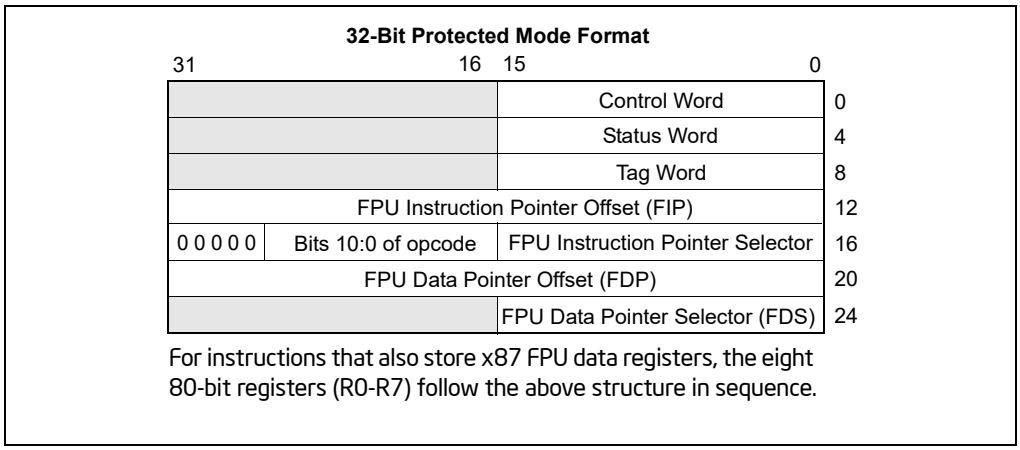


Figure 8-9. Protected Mode x87 FPU State Image in Memory, 32-Bit Format

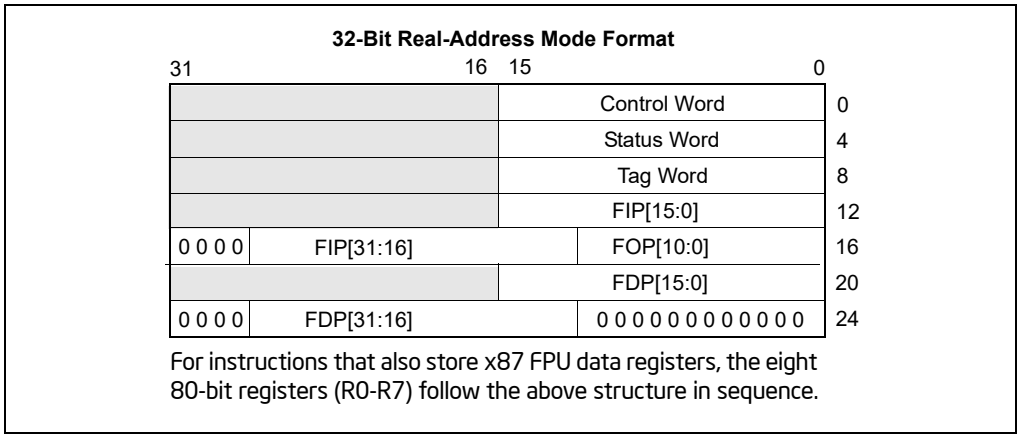


Figure 8-10. Real Mode x87 FPU State Image in Memory, 32-Bit Format

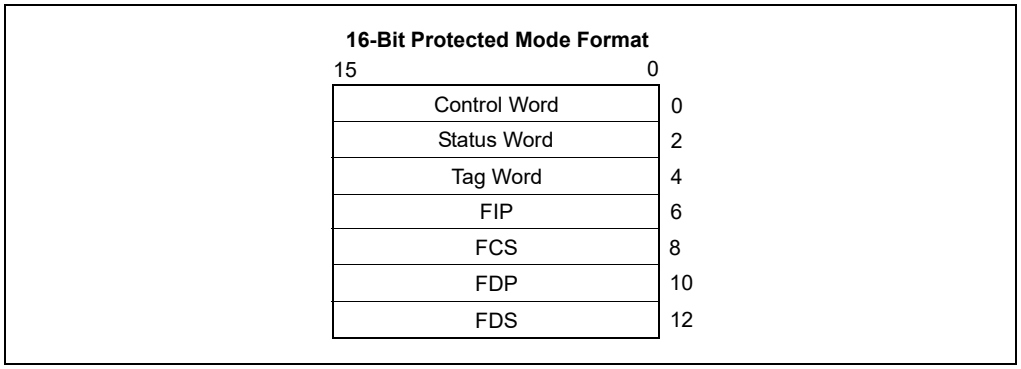


Figure 8-11. Protected Mode x87 FPU State Image in Memory, 16-Bit Format

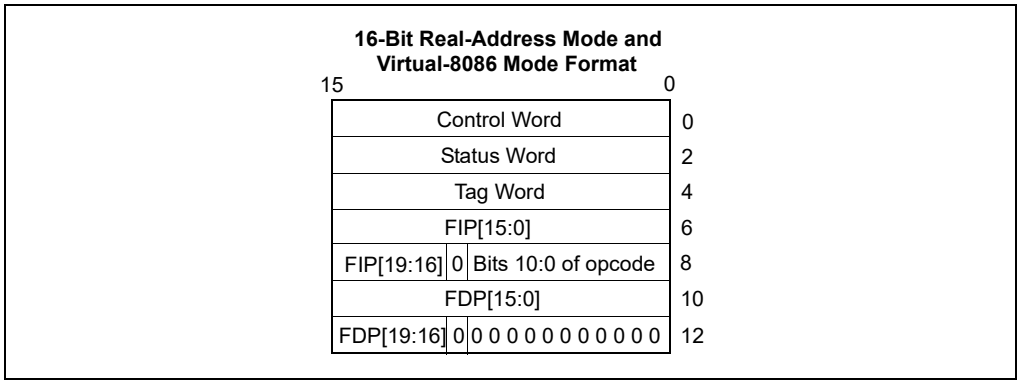


Figure 8-12. Real Mode x87 FPU State Image in Memory, 16-Bit Format

8.1.11 Saving the x87 FPU's State with FXSAVE

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

8.2 X87 FPU DATA TYPES

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

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

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

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

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

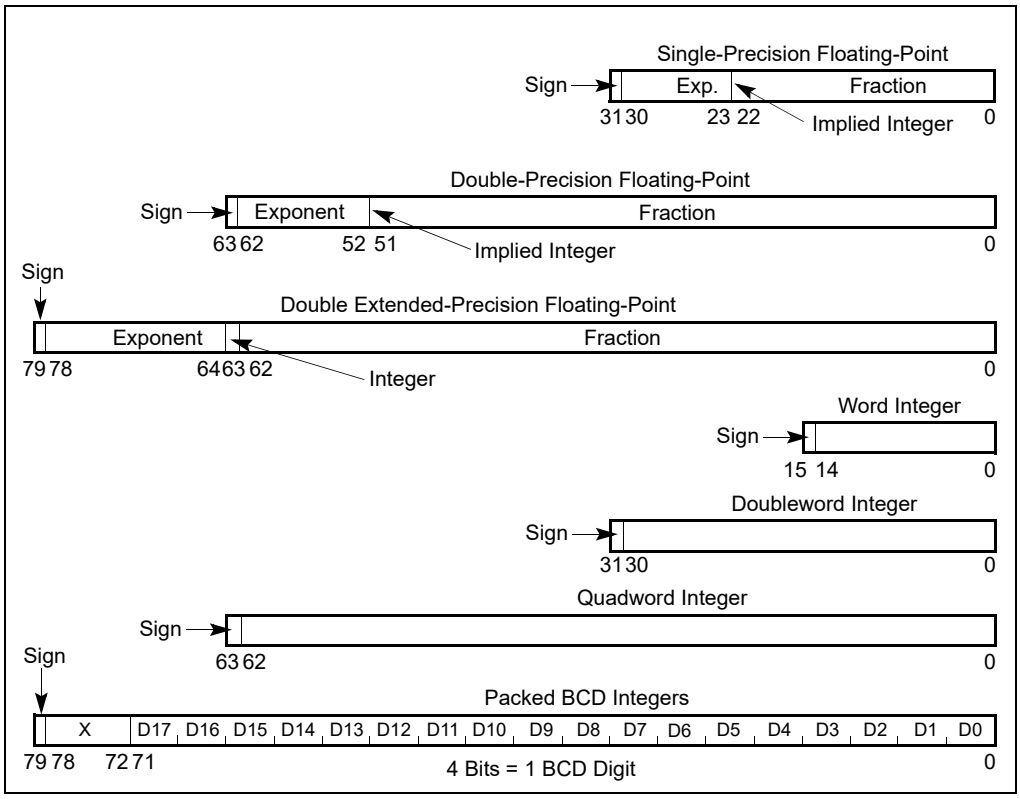


Figure 8-13. x87 FPU Data Type Formats

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

8.2.1 Indefinites

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

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

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

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

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

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

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

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

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

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

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

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

8.3 X87 FPU INSTRUCTION SET

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

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

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

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

8.3.1 Escape (ESC) Instructions

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

8.3.2 x87 FPU Instruction Operands

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

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

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

8.3.3 Data Transfer Instructions

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

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

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

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

Table 8-4. Data Transfer Instructions

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

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

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

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

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

Table 8-5. Floating-Point Conditional Move Instructions

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

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

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

8.3.4 Load Constant Instructions

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

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

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

8.3.5 Basic Arithmetic Instructions

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

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

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

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

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

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

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

FSUB:

$ST(0) \leftarrow ST(0) - ST(i)$
$ST(i) \leftarrow ST(i) - ST(0)$

FSUBR:

$ST(0) \leftarrow ST(i) - ST(0)$
$ST(i) \leftarrow ST(0) - ST(i)$

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

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

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

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

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

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

8.3.6 Comparison and Classification Instructions

The following instructions compare or classify floating-point values:

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

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

FICOM/FICOMP Compare integer and set x87 FPU condition code flags.

FCOMI/FCOMIP Compare floating point and set EFLAGS status flags.

FUCOMI/FUCOMIP Unordered compare floating point and set EFLAGS status flags.

FTST Test (compare floating point with 0.0).

FXAM Examine.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3.6.1 Branching on the x87 FPU Condition Codes

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

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

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

Table 8-8. TEST Instruction Constants for Conditional Branching

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

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

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

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

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

8.3.7 Trigonometric Instructions

The following instructions perform four common trigonometric functions:

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

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

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

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

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

8.3.8 Approximation of Pi

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

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

$$Pi = 0.f * 2^2$$

where the fraction f is expressed in binary form as:

f = C90FDAA2 2168C234 C

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

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

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

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

8.3.9 Logarithmic, Exponential, and Scale

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

FYL2X	Logarithm
FYL2XP1	Logarithm epsilon
F2XM1	Exponential
FSCALE	Scale

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

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

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

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

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

8.3.10 Transcendental Instruction Accuracy

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

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

where *k* is an integer such that:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3.11 x87 FPU Control Instructions

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

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

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

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

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

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

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

8.3.12 Waiting vs. Non-waiting Instructions

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

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

NOTES

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

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

8.3.13 Unsupported x87 FPU Instructions

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

8.4 X87 FPU FLOATING-POINT EXCEPTION HANDLING

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

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

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

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

NOTE

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

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

8.4.1 Arithmetic vs. Non-arithmetic Instructions

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

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

Table 8-9. Arithmetic and Non-arithmetic Instructions

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

NOTE:

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

8.5 X87 FPU FLOATING-POINT EXCEPTION CONDITIONS

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

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

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

8.5.1 Invalid Operation Exception

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

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

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

8.5.1.1 Stack Overflow or Underflow Exception (#IS)

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

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

NOTES

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

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

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

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

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

8.5.1.2 Invalid Arithmetic Operand Exception (#IA)

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

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

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

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

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

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

8.5.2 Denormal Operand Exception (#D)

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

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

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

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

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

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

8.5.3 Divide-By-Zero Exception (#Z)

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

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

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

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

8.5.4 Numeric Overflow Exception (#O)

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

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

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

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

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

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

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

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

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

8.5.5 Numeric Underflow Exception (#U)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.6 X87 FPU EXCEPTION SYNCHRONIZATION

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

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

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

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

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

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

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

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

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

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

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

8.7 HANDLING X87 FPU EXCEPTIONS IN SOFTWARE

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

8.7.1 Native Mode

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

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

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

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

8.7.2 MS-DOS* Compatibility Sub-mode

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

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

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

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

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

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

8.7.3 Handling x87 FPU Exceptions in Software

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

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

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

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

5. Updates to Chapter 11, Volume 1

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

Changes to this chapter: Update to Section 11.4.4.3 "Memory Ordering Instructions".

CHAPTER 11
PROGRAMMING WITH INTEL® STREAMING SIMD EXTENSIONS 2 (INTEL® SSE2)

The streaming SIMD extensions 2 (SSE2) were introduced into the IA-32 architecture in the Pentium 4 and Intel Xeon processors. These extensions enhance the performance of IA-32 processors for advanced 3-D graphics, video decoding/encoding, speech recognition, E-commerce, Internet, scientific, and engineering applications.

This chapter describes the SSE2 extensions and provides information to assist in writing application programs that use these and the SSE extensions.

11.1 OVERVIEW OF SSE2 EXTENSIONS

SSE2 extensions use the single instruction multiple data (SIMD) execution model that is used with MMX technology and SSE extensions. They extend this model with support for packed double-precision floating-point values and for 128-bit packed integers.

If CPUID.01H:EDX.SSE2[bit 26] = 1, SSE2 extensions are present.

SSE2 extensions add the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications and operating systems.

- Six data types:
  - 128-bit packed double-precision floating-point (two IEEE Standard 754 double-precision floating-point values packed into a double quadword)
  - 128-bit packed byte integers
  - 128-bit packed word integers
  - 128-bit packed doubleword integers
  - 128-bit packed quadword integers
- Instructions to support the additional data types and extend existing SIMD integer operations:
  - Packed and scalar double-precision floating-point instructions
  - Additional 64-bit and 128-bit SIMD integer instructions
  - 128-bit versions of SIMD integer instructions introduced with the MMX technology and the SSE extensions
  - Additional cacheability-control and instruction-ordering instructions
- Modifications to existing IA-32 instructions to support SSE2 features:
  - Extensions and modifications to the CPUID instruction
  - Modifications to the RDPMC instruction

These new features extend the IA-32 architecture's SIMD programming model in three important ways:

- They provide the ability to perform SIMD operations on pairs of packed double-precision floating-point values. This permits higher precision computations to be carried out in XMM registers, which enhances processor performance in scientific and engineering applications and in applications that use advanced 3-D geometry techniques (such as ray tracing). Additional flexibility is provided with instructions that operate on single (scalar) double-precision floating-point values located in the low quadword of an XMM register.
- They provide the ability to operate on 128-bit packed integers (bytes, words, doublewords, and quadwords) in XMM registers. This provides greater flexibility and greater throughput when performing SIMD operations on packed integers. The capability is particularly useful for applications such as RSA authentication and RC5 encryption. Using the full set of SIMD registers, data types, and instructions provided with the MMX technology and SSE/SSE2 extensions, programmers can develop algorithms that finely mix packed single- and double-precision floating-point data and 64- and 128-bit packed integer data.
- SSE2 extensions enhance the support introduced with SSE extensions for controlling the cacheability of SIMD data. SSE2 cache control instructions provide the ability to stream data in and out of the XMM registers without polluting the caches and the ability to prefetch data before it is actually used.

SSE2 extensions are fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate SSE2 extensions, as well as in the presence of applications that incorporate these extensions. Enhancements to the CPUID instruction permit detection of the SSE2 extensions. Also, because the SSE2 extensions use the same registers as the SSE extensions, no new operating-system support is required for saving and restoring program state during a context switch beyond that provided for the SSE extensions.

SSE2 extensions are accessible from all IA-32 execution modes: protected mode, real address mode, virtual 8086 mode.

The following sections in this chapter describe the programming environment for SSE2 extensions including: the 128-bit XMM floating-point register set, data types, and SSE2 instructions. It also describes exceptions that can be generated with the SSE and SSE2 instructions and gives guidelines for writing applications with SSE and SSE2 extensions.

For additional information about SSE2 extensions, see:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B,* provide a detailed description of individual SSE3 instructions.
- Chapter 13, "System Programming for Instruction Set Extensions and Processor Extended States," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* gives guidelines for integrating the SSE and SSE2 extensions into an operating-system environment.

11.2 SSE2 PROGRAMMING ENVIRONMENT

Figure 11-1 shows the programming environment for SSE2 extensions. No new registers or other instruction execution state are defined with SSE2 extensions. SSE2 instructions use the XMM registers, the MMX registers, and/or IA-32 general-purpose registers, as follows:

- **XMM registers —** These eight registers (see Figure 10-2) are used to operate on packed or scalar double-precision floating-point data. Scalar operations are operations performed on individual (unpacked) double-precision floating-point values stored in the low quadword of an XMM register. XMM registers are also used to perform operations on 128-bit packed integer data. They are referenced by the names XMM0 through XMM7.

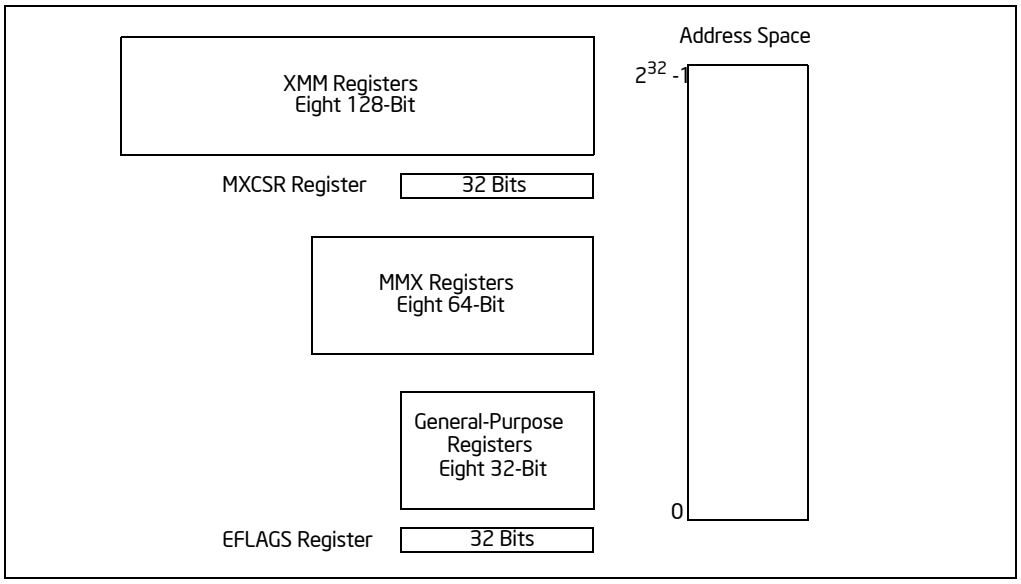


Figure 11-1. Steaming SIMD Extensions 2 Execution Environment

- **MXCSR register —** This 32-bit register (see Figure 10-3) provides status and control bits used in floating-point operations. The denormals-are-zeros and flush-to-zero flags in this register provide a higher performance alternative for the handling of denormal source operands and denormal (underflow) results. For more

information on the functions of these flags see Section 10.2.3.4, "Denormals-Are-Zeros," and Section 10.2.3.3, "Flush-To-Zero."

- **MMX registers —** These eight registers (see Figure 9-2) are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.
- **General-purpose registers —** The eight general-purpose registers (see Figure 3-5) are used along with the existing IA-32 addressing modes to address operands in memory. MMX and XMM registers cannot be used to address memory. The general-purpose registers are also used to hold operands for some SSE2 instructions. These registers are referenced by the names EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.
- **EFLAGS register —** This 32-bit register (see Figure 3-8) is used to record the results of some compare operations.

11.2.1 SSE2 in 64-Bit Mode and Compatibility Mode

In compatibility mode, SSE2 extensions function like they do in protected mode. In 64-bit mode, eight additional XMM registers are accessible. Registers XMM8-XMM15 are accessed by using REX prefixes.

Memory operands are specified using the ModR/M, SIB encoding described in Section 3.7.5.

Some SSE2 instructions may be used to operate on general-purpose registers. Use the REX.W prefix to access 64-bit general-purpose registers. Note that if a REX prefix is used when it has no meaning, the prefix is ignored.

11.2.2 Compatibility of SSE2 Extensions with SSE, MMX Technology and x87 FPU Programming Environment

SSE2 extensions do not introduce any new state to the IA-32 execution environment beyond that of SSE. SSE2 extensions represent an enhancement of SSE extensions; they are fully compatible and share the same state information. SSE and SSE2 instructions can be executed together in the same instruction stream without the need to save state when switching between instruction sets.

XMM registers are independent of the x87 FPU and MMX registers; so SSE and SSE2 operations performed on XMM registers can be performed in parallel with x87 FPU or MMX technology operations (see Section 11.6.7, "Interaction of SSE/SSE2 Instructions with x87 FPU and MMX Instructions").

The FXSAVE and FXRSTOR instructions save and restore the SSE and SSE2 states along with the x87 FPU and MMX states.

11.2.3 Denormals-Are-Zeros Flag

The denormals-are-zeros flag (bit 6 in the MXCSR register) was introduced into the IA-32 architecture with the SSE2 extensions. See Section 10.2.3.4, "Denormals-Are-Zeros," for a description of this flag.

11.3 SSE2 DATA TYPES

SSE2 extensions introduced one 128-bit packed floating-point data type and four 128-bit SIMD integer data types to the IA-32 architecture (see Figure 11-2).

- **Packed double-precision floating-point —** This 128-bit data type consists of two IEEE 64-bit double-precision floating-point values packed into a double quadword. (See Figure 4-3 for the layout of a 64-bit double-precision floating-point value; refer to Section 4.2.2, "Floating-Point Data Types," for a detailed description of double-precision floating-point values.)
- **128-bit packed integers —** The four 128-bit packed integer data types can contain 16 byte integers, 8 word integers, 4 doubleword integers, or 2 quadword integers. (Refer to Section 4.6.2, "128-Bit Packed SIMD Data Types," for a detailed description of the 128-bit packed integers.)

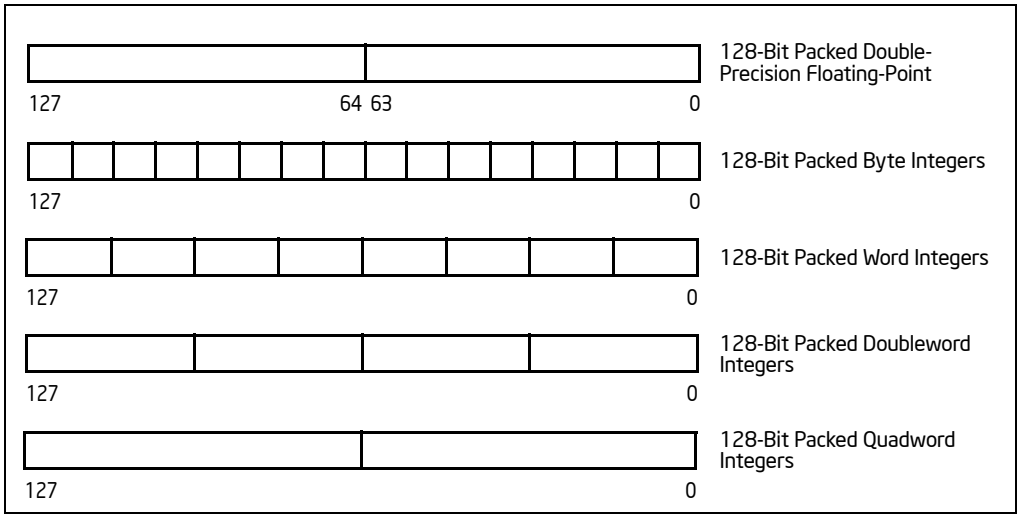


Figure 11-2. Data Types Introduced with the SSE2 Extensions

All of these data types are operated on in XMM registers or memory. Instructions are provided to convert between these 128-bit data types and the 64-bit and 32-bit data types.

The address of a 128-bit packed memory operand must be aligned on a 16-byte boundary, except in the following cases:

- a MOVUPD instruction which supports unaligned accesses
- scalar instructions that use an 8-byte memory operand that is not subject to alignment requirements

Figure 4-2 shows the byte order of 128-bit (double quadword) and 64-bit (quadword) data types in memory.

11.4 SSE2 INSTRUCTIONS

The SSE2 instructions are divided into four functional groups:

- Packed and scalar double-precision floating-point instructions
- 64-bit and 128-bit SIMD integer instructions
- 128-bit extensions of SIMD integer instructions introduced with the MMX technology and the SSE extensions
- Cacheability-control and instruction-ordering instructions

The following sections provide more information about each group.

11.4.1 Packed and Scalar Double-Precision Floating-Point Instructions

The packed and scalar double-precision floating-point instructions are divided into the following sub-groups:

- Data movement instructions
- Arithmetic instructions
- Comparison instructions
- Conversion instructions
- Logical instructions
- Shuffle instructions

The packed double-precision floating-point instructions perform SIMD operations similarly to the packed single-precision floating-point instructions (see Figure 11-3). Each source operand contains two double-precision floating-

point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, and X1 and Y1) in each operand.

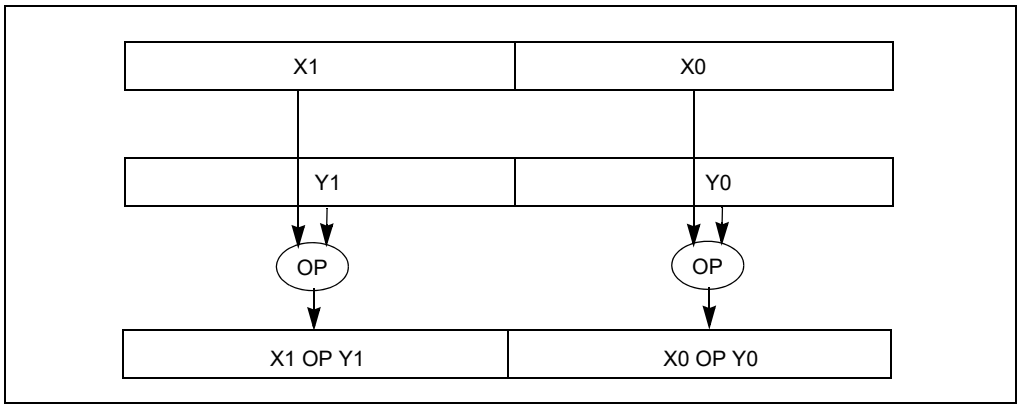


Figure 11-3. Packed Double-Precision Floating-Point Operations

The scalar double-precision floating-point instructions operate on the low (least significant) quadwords of two source operands (X0 and Y0), as shown in Figure 11-4. The high quadword (X1) of the first source operand is passed through to the destination. The scalar operations are similar to the floating-point operations performed in x87 FPU data registers with the precision control field in the x87 FPU control word set for double precision (53-bit significand), except that x87 stack operations use a 15-bit exponent range for the result while SSE2 operations use an 11-bit exponent range.

See Section 11.6.8, "Compatibility of SIMD and x87 FPU Floating-Point Data Types," for more information about obtaining compatible results when performing both scalar double-precision floating-point operations in XMM registers and in x87 FPU data registers.

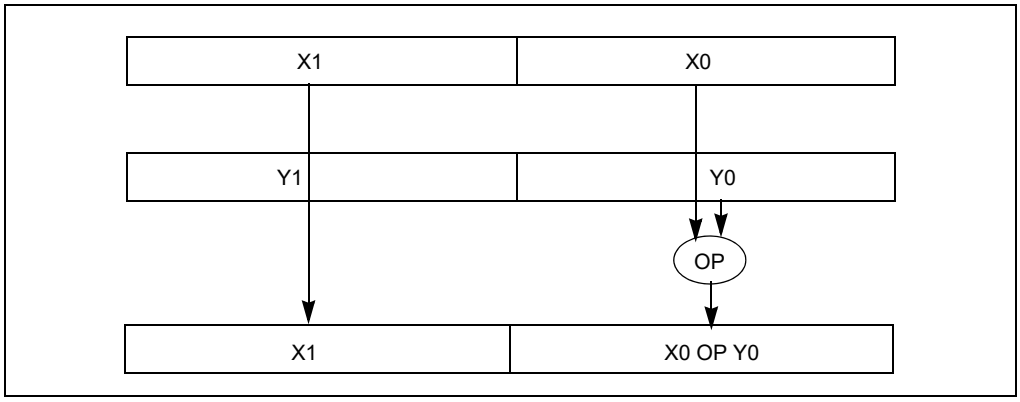


Figure 11-4. Scalar Double-Precision Floating-Point Operations

11.4.1.1 Data Movement Instructions

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

The MOVAPD (move aligned packed double-precision floating-point) instruction transfers a 128-bit packed double-precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. The memory address must be aligned to a 16-byte boundary; if not, a general-protection exception (GP#) is generated.

The MOVUPD (move unaligned packed double-precision floating-point) instruction transfers a 128-bit packed double-precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required.

The MOVSD (move scalar double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the low quadword of an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVHPD (move high packed double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the high quadword of an XMM register or vice versa. The low quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVLPD (move low packed double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the low quadword of an XMM register or vice versa. The high quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVMSKPD (move packed double-precision floating-point mask) instruction extracts the sign bit of each of the two packed double-precision floating-point numbers in an XMM register and saves them in a general-purpose register. This 2-bit value can then be used as a condition to perform branching.

11.4.1.2 SSE2 Arithmetic Instructions

SSE2 arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double-precision floating-point values.

The ADDPD (add packed double-precision floating-point values) and SUBPD (subtract packed double-precision floating-point values) instructions add and subtract, respectively, two packed double-precision floating-point operands.

The ADDSD (add scalar double-precision floating-point values) and SUBSD (subtract scalar double-precision floating-point values) instructions add and subtract, respectively, the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The MULPD (multiply packed double-precision floating-point values) instruction multiplies two packed double-precision floating-point operands.

The MULSD (multiply scalar double-precision floating-point values) instruction multiplies the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The DIVPD (divide packed double-precision floating-point values) instruction divides two packed double-precision floating-point operands.

The DIVSD (divide scalar double-precision floating-point values) instruction divides the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The SQRTPD (compute square roots of packed double-precision floating-point values) instruction computes the square roots of the values in a packed double-precision floating-point operand.

The SQRTSD (compute square root of scalar double-precision floating-point values) instruction computes the square root of the low double-precision floating-point value in the source operand and stores the result in the low quadword of the destination operand.

The MAXPD (return maximum of packed double-precision floating-point values) instruction compares the corresponding values in two packed double-precision floating-point operands and returns the numerically greater value from each comparison to the destination operand.

The MAXSD (return maximum of scalar double-precision floating-point values) instruction compares the low double-precision floating-point values from two packed double-precision floating-point operands and returns the numerically higher value from the comparison to the low quadword of the destination operand.

The MINPD (return minimum of packed double-precision floating-point values) instruction compares the corresponding values from two packed double-precision floating-point operands and returns the numerically lesser value from each comparison to the destination operand.

The MINSD (return minimum of scalar double-precision floating-point values) instruction compares the low values from two packed double-precision floating-point operands and returns the numerically lesser value from the comparison to the low quadword of the destination operand.

11.4.1.3 SSE2 Logical Instructions

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

The ANDPD (bitwise logical AND of packed double-precision floating-point values) instruction returns the logical AND of two packed double-precision floating-point operands.

The ANDNPD (bitwise logical AND NOT of packed double-precision floating-point values) instruction returns the logical AND NOT of two packed double-precision floating-point operands.

The ORPD (bitwise logical OR of packed double-precision floating-point values) instruction returns the logical OR of two packed double-precision floating-point operands.

The XORPD (bitwise logical XOR of packed double-precision floating-point values) instruction returns the logical XOR of two packed double-precision floating-point operands.

11.4.1.4 SSE2 Comparison 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.

The CMPPD (compare packed double-precision floating-point values) instruction compares the corresponding values from two packed double-precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for each comparison to the destination operand. The value of the immediate operand allows the selection of any of eight compare conditions: equal, less than, less than equal, unordered, not equal, not less than, not less than or equal, or ordered.

The CMPSD (compare scalar double-precision floating-point values) instruction compares the low values from two packed double-precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for the comparison to the low quadword of the destination operand. The immediate operand selects the compare condition as with the CMPPD instruction.

The COMISD (compare scalar double-precision floating-point values and set EFLAGS) and UCOMISD (unordered compare scalar double-precision floating-point values and set EFLAGS) instructions compare the low values of two packed double-precision floating-point operands and set the ZF, PF, and CF flags in the EFLAGS register to show the result (greater than, less than, equal, or unordered). These two instructions differ as follows: the COMISD instruction signals a floating-point invalid-operation (#I) exception when a source operand is either a QNaN or an SNaN; the UCOMISD instruction only signals an invalid-operation exception when a source operand is an SNaN.

11.4.1.5 SSE2 Shuffle and Unpack Instructions

SSE2 shuffle instructions shuffle the contents of two packed double-precision floating-point values and store the results in the destination operand.

The SHUFPD (shuffle packed double-precision floating-point values) instruction places either of the two packed double-precision floating-point values from the destination operand in the low quadword of the destination operand, and places either of the two packed double-precision floating-point values from source operand in the high quadword of the destination operand (see Figure 11-5). By using the same register for the source and destination operands, the SHUFPD instruction can swap two packed double-precision floating-point values.

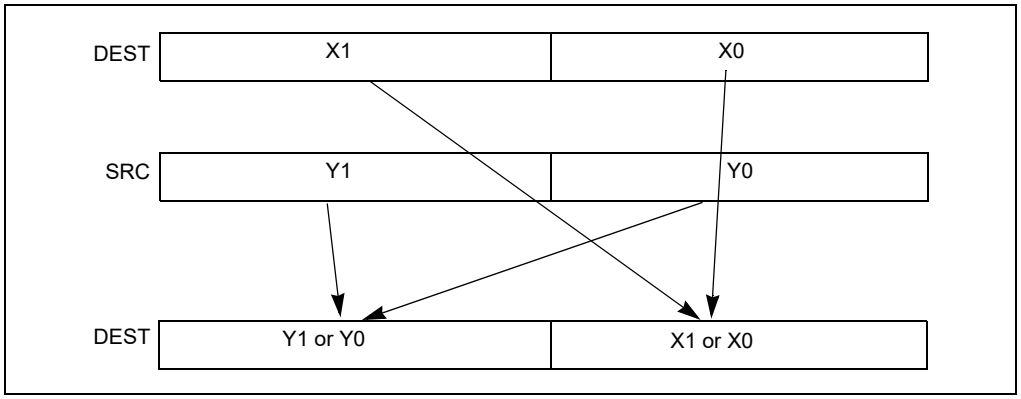


Figure 11-5. SHUFPD Instruction, Packed Shuffle Operation

The UNPCKHPD (unpack and interleave high packed double-precision floating-point values) instruction performs an interleaved unpack of the high values from the source and destination operands and stores the result in the destination operand (see Figure 11-6).

The UNPCKLPD (unpack and interleave low packed double-precision floating-point values) instruction performs an interleaved unpack of the low values from the source and destination operands and stores the result in the destination operand (see Figure 11-7).

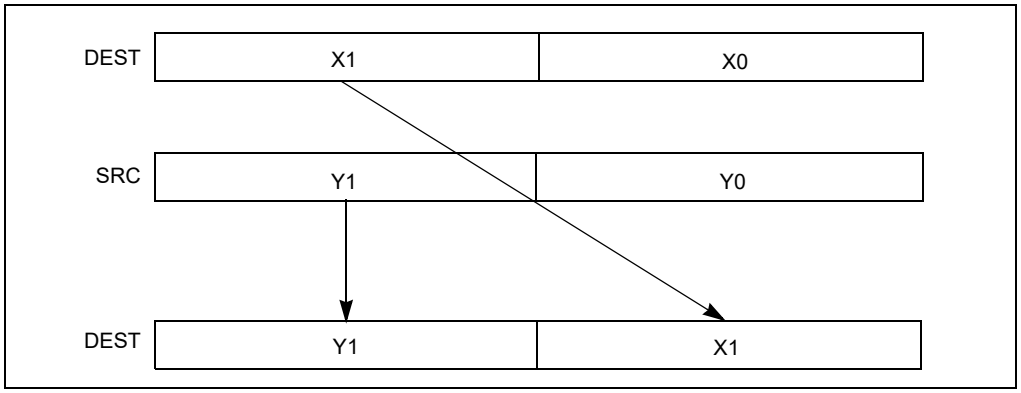


Figure 11-6. UNPCKHPD Instruction, High Unpack and Interleave Operation

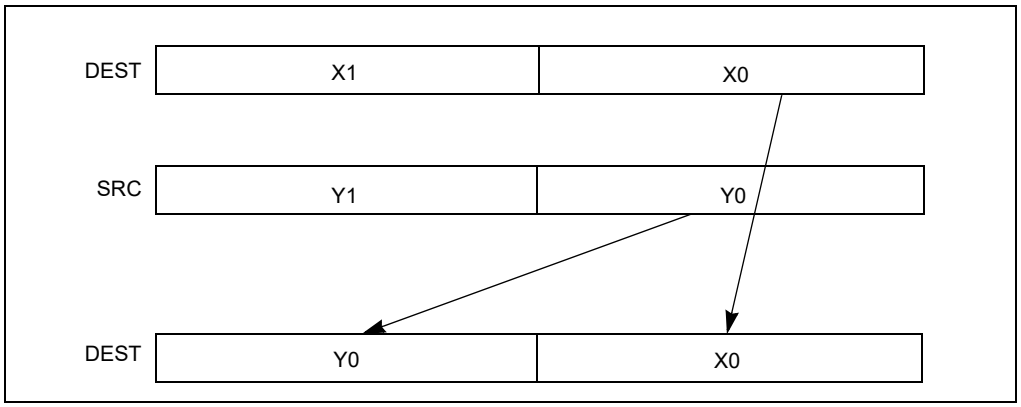


Figure 11-7. UNPCKLPD Instruction, Low Unpack and Interleave Operation

11.4.1.6 SSE2 Conversion Instructions

SSE2 conversion instructions (see Figure 11-8) support packed and scalar conversions between:

- Double-precision and single-precision floating-point formats
- Double-precision floating-point and doubleword integer formats
- Single-precision floating-point and doubleword integer formats

Conversion between double-precision and single-precision floating-points values — The following instructions convert operands between double-precision and single-precision floating-point formats. The operands being operated on are contained in XMM registers or memory (at most, one operand can reside in memory; the destination is always an MMX register).

The CVTPS2PD (convert packed single-precision floating-point values to packed double-precision floating-point values) instruction converts two packed single-
precision floating-point values to two double-precision floating-point values.

The CVTPD2PS (convert packed double-precision floating-point values to packed single-precision floating-point values) instruction converts two packed double-
precision floating-point values to two single-precision floating-point values. When a conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTSS2SD (convert scalar single-precision floating-point value to scalar double-precision floating-point value) instruction converts a single-precision floating-point value to a double-precision floating-point value.

The CVTSD2SS (convert scalar double-precision floating-point value to scalar single-precision floating-point value) instruction converts a double-precision floating-point value to a single-precision floating-point value. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

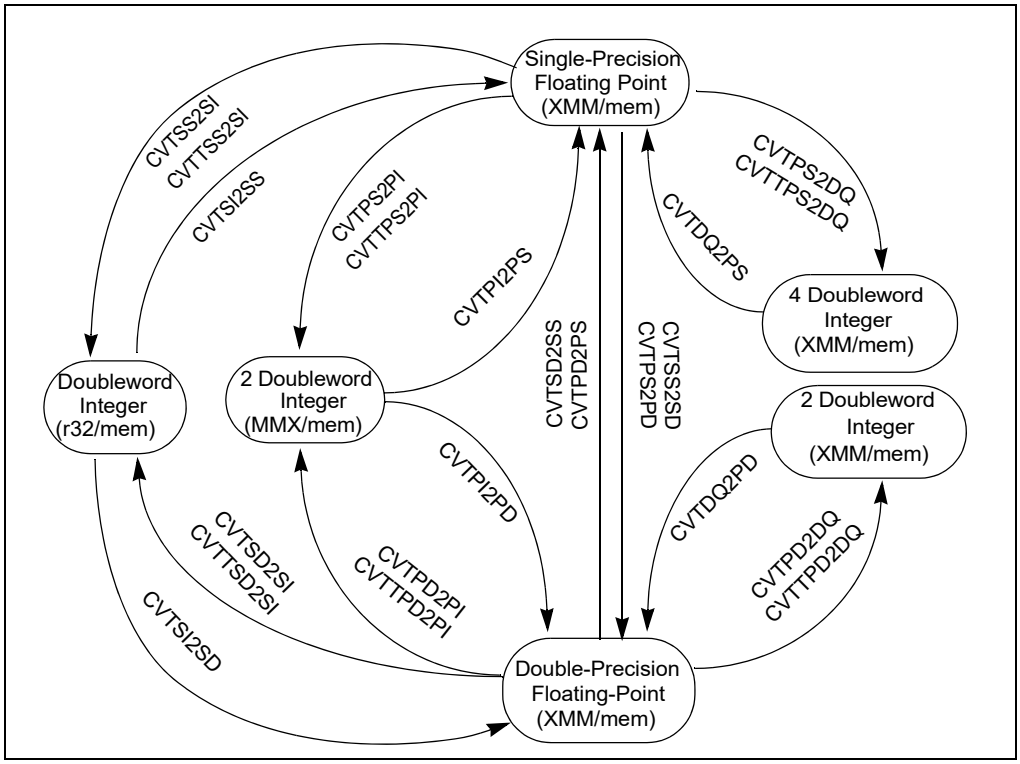


Figure 11-8. SSE and SSE2 Conversion Instructions

Conversion between double-precision floating-point values and doubleword integers — The following instructions convert operands between double-precision floating-point and doubleword integer formats. Operands

are housed in XMM registers, MMX registers, general registers or memory (at most one operand can reside in memory; the destination is always an XMM, MMX, or general register).

The CVTPD2PI (convert packed double-precision floating-point values to packed doubleword integers) instruction converts two packed double-precision floating-point numbers to two packed signed doubleword integers, with the result stored in an MMX register. When rounding to an integer value, the source value is rounded according to the rounding mode in the MXCSR register. The CVTTPD2PI (convert with truncation packed double-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPD2PI instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2, "Truncation with SSE and SSE2 Conversion Instructions").

The CVTPI2PD (convert packed doubleword integers to packed double-precision floating-point values) instruction converts two packed signed doubleword integers to two double-precision floating-point values.

The CVTPD2DQ (convert packed double-precision floating-point values to packed doubleword integers) instruction converts two packed double-precision floating-point numbers to two packed signed doubleword integers, with the result stored in the low quadword of an XMM register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTTPD2DQ (convert with truncation packed double-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPD2DQ instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2, "Truncation with SSE and SSE2 Conversion Instructions").

The CVTDQ2PD (convert packed doubleword integers to packed double-precision floating-point values) instruction converts two packed signed doubleword integers located in the low-order doublewords of an XMM register to two double-precision floating-point values.

The CVTSD2SI (convert scalar double-precision floating-point value to doubleword integer) instruction converts a double-precision floating-point value to a doubleword integer, and stores the result in a general-purpose register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTTSD2SI (convert with truncation scalar double-precision floating-point value to doubleword integer) instruction is similar to the CVTSD2SI instruction except that truncation is used to round the source value to an integer value (see Section 4.8.4.2, "Truncation with SSE and SSE2 Conversion Instructions").

The CVTSI2SD (convert doubleword integer to scalar double-precision floating-point value) instruction converts a signed doubleword integer in a general-purpose register to a double-precision floating-point number, and stores the result in an XMM register.

Conversion between single-precision floating-point and doubleword integer formats — These instructions convert between packed single-precision floating-point and packed doubleword integer formats. Operands are housed in XMM registers, MMX registers, general registers, or memory (the latter for at most one source operand). The destination is always an XMM, MMX, or general register. These SSE2 instructions supplement conversion instructions (CVTPI2PS, CVTPS2PI, CVTTPS2PI, CVTSI2SS, CVTSS2SI, and CVTTSS2SI) introduced with SSE extensions.

The CVTPS2DQ (convert packed single-precision floating-point values to packed doubleword integers) instruction converts four packed single-precision floating-point values to four packed signed doubleword integers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned. The CVTTPS2DQ (convert with truncation packed single-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPS2DQ instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2, "Truncation with SSE and SSE2 Conversion Instructions").

The CVTDQ2PS (convert packed doubleword integers to packed single-precision floating-point values) instruction converts four packed signed doubleword integers to four packed single-precision floating-point numbers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned.

11.4.2 SSE2 64-Bit and 128-Bit SIMD Integer Instructions

SSE2 extensions add several 128-bit packed integer instructions to the IA-32 architecture. Where appropriate, a 64-bit version of each of these instructions is also provided. The 128-bit versions of instructions operate on data in XMM registers; 64-bit versions operate on data in MMX registers. The instructions follow.

The MOVDQA (move aligned double quadword) instruction transfers a double quadword operand from memory to an XMM register or vice versa; or between XMM registers. The memory address must be aligned to a 16-byte boundary; otherwise, a general-protection exception (#GP) is generated.

The MOVDQU (move unaligned double quadword) instruction performs the same operations as the MOVDQA instruction, except that 16-byte alignment of a memory address is not required.

The PADDQ (packed quadword add) instruction adds two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. This instruction can operate on either unsigned or signed (two's complement notation) integer operands.

The PSUBQ (packed quadword subtract) instruction subtracts two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. Like the PADDQ instruction, PSUBQ can operate on either unsigned or signed (two's complement notation) integer operands.

The PMULUDQ (multiply packed unsigned doubleword integers) instruction performs an unsigned multiply of unsigned doubleword integers and returns a quadword result. Both 64-bit and 128-bit versions of this instruction are available. The 64-bit version operates on two doubleword integers stored in the low doubleword of each source operand, and the quadword result is returned to an MMX register. The 128-bit version performs a packed multiply of two pairs of doubleword integers. Here, the doublewords are packed in the first and third doublewords of the source operands, and the quadword results are stored in the low and high quadwords of an XMM register.

The PSHUFLW (shuffle packed low words) instruction shuffles the word integers packed into the low quadword of the source operand and stores the shuffled result in the low quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFHW (shuffle packed high words) instruction shuffles the word integers packed into the high quadword of the source operand and stores the shuffled result in the high quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFD (shuffle packed doubleword integers) instruction shuffles the doubleword integers packed into the source operand and stores the shuffled result in the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSLLDQ (shift double quadword left logical) instruction shifts the contents of the source operand to the left by the amount of bytes specified by an immediate operand. The empty low-order bytes are cleared (set to 0).

The PSRLDQ (shift double quadword right logical) instruction shifts the contents of the source operand to the right by the amount of bytes specified by an immediate operand. The empty high-order bytes are cleared (set to 0).

The PUNPCKHQDQ (Unpack high quadwords) instruction interleaves the high quadword of the source operand and the high quadword of the destination operand and writes them to the destination register.

The PUNPCKLQDQ (Unpack low quadwords) instruction interleaves the low quadwords of the source operand and the low quadwords of the destination operand and writes them to the destination register.

Two additional SSE instructions enable data movement from the MMX registers to the XMM registers.

The MOVQ2DQ (move quadword integer from MMX to XMM registers) instruction moves the quadword integer from an MMX source register to an XMM destination register.

The MOVDQ2Q (move quadword integer from XMM to MMX registers) instruction moves the low quadword integer from an XMM source register to an MMX destination register.

11.4.3 128-Bit SIMD Integer Instruction Extensions

All of 64-bit SIMD integer instructions introduced with MMX technology and SSE extensions (with the exception of the PSHUFW instruction) have been extended by SSE2 extensions to operate on 128-bit packed integer operands located in XMM registers. The 128-bit versions of these instructions follow the same SIMD conventions regarding packed operands as the 64-bit versions. For example, where the 64-bit version of the PADDB instruction operates on 8 packed bytes, the 128-bit version operates on 16 packed bytes.

11.4.4 Cacheability Control and Memory Ordering Instructions

SSE2 extensions that give programs more control over the caching, loading, and storing of data. are described below.

11.4.4.1 FLUSH Cache Line

The CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor's cache hierarchy, and it is broadcast throughout the cache coherency domain.

NOTE

CLFLUSH was introduced with the SSE2 extensions. However, the instruction can be implemented in IA-32 processors that do not implement the SSE2 extensions. Detect CLFLUSH using the feature bit (if CPUID.01H:EDX.CLFSH[bit 19] = 1).

11.4.4.2 Cacheability Control Instructions

The following four instructions enable data from XMM and general-purpose registers to be stored to memory using a non-temporal hint. The non-temporal hint directs the processor to store data to memory without writing the data into the cache hierarchy. See Section 10.4.6.2, "Caching of Temporal vs. Non-Temporal Data," for more information about non-temporal stores and hints.

The MOVNTDQ (store double quadword using non-temporal hint) instruction stores packed integer data from an XMM register to memory, using a non-temporal hint.

The MOVNTPD (store packed double-precision floating-point values using non-temporal hint) instruction stores packed double-precision floating-point data from an XMM register to memory, using a non-temporal hint.

The MOVNTI (store doubleword using non-temporal hint) instruction stores integer data from a general-purpose register to memory, using a non-temporal hint.

The MASKMOVDQU (store selected bytes of double quadword) instruction stores selected byte integers from an XMM register to memory, using a byte mask to selectively write the individual bytes. The memory location does not need to be aligned on a natural boundary. This instruction also uses a non-temporal hint.

11.4.4.3 Memory Ordering Instructions

SSE2 extensions introduce two new fence instructions (LFENCE and MFENCE) as companions to the SFENCE instruction introduced with SSE extensions.

The LFENCE instruction establishes a memory fence for loads. It guarantees ordering between two loads and prevents speculative loads from passing the load fence (that is, no speculative loads are allowed until all loads specified before the load fence have been carried out).

The MFENCE instruction establishes a memory fence for both loads and stores. The processor ensures that no load or store after MFENCE will become globally visible until all loads and stores before MFENCE are globally visible.[1] Note that the sequences LFENCE;SFENCE and SFENCE;LFENCE are not equivalent to MFENCE because neither ensures that older stores are globally observed prior to younger loads.

11.4.4.4 Pause

The PAUSE instruction is provided to improve the performance of "spin-wait loops" executed on a Pentium 4 or Intel Xeon processor. On a Pentium 4 processor, it also provides the added benefit of reducing processor power consumption while executing a spin-wait loop. It is recommended that a PAUSE instruction always be included in the code sequence for a spin-wait loop.

1. A load is considered to become globally visible when the value to be loaded is determined.

11.4.5 Branch Hints

SSE2 extensions designate two instruction prefixes (2EH and 3EH) to provide branch hints to the processor (see "Instruction Prefixes" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A)*. These prefixes can only be used with the Jcc instruction and only at the machine code level (that is, there are no mnemonics for the branch hints).

11.5 SSE, SSE2, AND SSE3 EXCEPTIONS

SSE/SSE2/SSE3 extensions generate two general types of exceptions:

- Non-numeric exceptions
- SIMD floating-point exceptions[1]

SSE/SSE2/SSE3 instructions can generate the same type of memory-access and non-numeric exceptions as other IA-32 architecture instructions. Existing exception handlers can generally handle these exceptions without any code modification. See "Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE, SSE2 and SSE3 Instructions" in Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* for a list of the non-numeric exceptions that can be generated by SSE/SSE2/SSE3 instructions and for guidelines for handling these exceptions.

SSE/SSE2/SSE3 instructions do not generate numeric exceptions on packed integer operations; however, they can generate numeric (SIMD floating-point) exceptions on packed single-precision and double-precision floating-point operations. These SIMD floating-point exceptions are defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic and are the same exceptions that are generated for x87 FPU instructions. See Section 11.5.1, "SIMD Floating-Point Exceptions," for a description of these exceptions.

11.5.1 SIMD Floating-Point Exceptions

SIMD floating-point exceptions are those exceptions that can be generated by SSE/SSE2/SSE3 instructions that operate on packed or scalar floating-point operands.

Six classes of SIMD floating-point exceptions can be generated:

- Invalid operation (#I)
- Divide-by-zero (#Z)
- Denormal operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (Precision) (#P)

All of these exceptions (except the denormal operand exception) are defined in IEEE Standard 754, and they are the same exceptions that are generated with the x87 floating-point instructions. Section 4.9, "Overview of Floating-Point Exceptions," gives a detailed description of these exceptions and of how and when they are generated. The following sections discuss the implementation of these exceptions in SSE/SSE2/SSE3 extensions.

All SIMD floating-point exceptions are precise and occur as soon as the instruction completes execution.

Each of the six exception conditions has a corresponding flag (IE, DE, ZE, OE, UE, and PE) and mask bit (IM, DM, ZM, OM, UM, and PM) in the MXCSR register (see Figure 10-3). The mask bits can be set with the LDMXCSR or FXRSTOR instruction; the mask and flag bits can be read with the STMXCSR or FXSAVE instruction.

The OSXMMEXCEPT flag (bit 10) of control register CR4 provides additional control over generation of SIMD floating-point exceptions by allowing the operating system to indicate whether or not it supports software exception handlers for SIMD floating-point exceptions. If an unmasked SIMD floating-point exception is generated and the OSXMMEXCEPT flag is set, the processor invokes a software exception handler by generating a SIMD floating-

1. The FISTTP instruction in SSE3 does not generate SIMD floating-point exceptions, but it can generate x87 FPU floating-point exceptions.

point exception (#XM). If the OSXMMEXCEPT bit is clear, the processor generates an invalid-opcode exception (#UD) on the first SSE or SSE2 instruction that detects a SIMD floating-point exception condition. See Section 11.6.2, "Checking for SSE/SSE2 Support."

11.5.2 SIMD Floating-Point Exception Conditions

The following sections describe the conditions that cause a SIMD floating-point exception to be generated and the masked response of the processor when these conditions are detected.

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

11.5.2.1 Invalid Operation Exception (#I)

The floating-point invalid-operation exception (#I) occurs in response to an invalid arithmetic operand. The flag (IE) and mask (IM) bits for the invalid operation exception are bits 0 and 7, respectively, in the MXCSR register.

If the invalid-operation exception is masked, the processor returns a QNaN, QNaN floating-point indefinite, integer indefinite, one of the source operands to the destination operand, or it sets the EFLAGS, depending on the operation being performed. When a value is returned to the destination operand, it overwrites the destination register specified by the instruction. Table 11-1 lists the invalid-arithmetic operations that the processor detects for instructions and the masked responses to these operations.

Table 11-1. Masked Responses of SSE/SSE2/SSE3 Instructions to Invalid Arithmetic Operations

Condition	Masked Response
ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, ADDSUBPD, ADDSUBPD, HADDPD, HADDPS, HSUBPD or HSUBPS instruction with an SNaN operand	Return the SNaN converted to a QNaN; Refer to Table 4-7 for more details
SQRTPS, SQRTSS, SQRTPD, or SQRTSD with SNaN operands	Return the SNaN converted to a QNaN
SQRTPS, SQRTSS, SQRTPD, or SQRTSD with negative operands (except zero)	Return the QNaN floating-point Indefinite
MAXPS, MAXSS, MAXPD, MAXSD, MINPS, MINSS, MINPD, or MINSD instruction with QNaN or SNaN operands	Return the source 2 operand value
CMPPS, CMPSS, CMPPD or CMPSD instruction with QNaN or SNaN operands	Return a mask of all 0s (except for the predicates "not-equal," "unordered," "not-less-than," or "not-less-than-or-equal," which returns a mask of all 1s)
CVTPD2PS, CVTSD2SS, CVTPS2PD, CVTSS2SD with SNaN operands	Return the SNaN converted to a QNaN
COMISS or COMISD with QNaN or SNaN operand(s)	Set EFLAGS values to "not comparable"
Addition of opposite signed infinities or subtraction of like-signed infinities	Return the QNaN floating-point Indefinite
Multiplication of infinity by zero	Return the QNaN floating-point Indefinite
Divide of (0/0) or (∞ / ∞)	Return the QNaN floating-point Indefinite
Conversion to integer when the value in the source register is a NaN, ∞, or exceeds the representable range for CVTPS2PI, CVTTPS2PI, CVTSS2SI, CVTTSS2SI, CVTPD2PI, CVTSD2SI, CVTPD2DQ, CVTTPD2PI, CVTTSD2SI, CVTTPD2DQ, CVTPS2DQ, or CVTTPS2DQ	Return the integer Indefinite

If the invalid operation exception is not masked, a software exception handler is invoked and the operands remain unchanged. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software."

Normally, when one or more of the source operands are QNaNs (and neither is an SNaN or in an unsupported format), an invalid-operation exception is not generated. The following instructions are exceptions to this rule: the COMISS and COMISD instructions; and the CMPPS, CMPSS, CMPPD, and CMPSD instructions (when the predicate is less than, less-than or equal, not less-than, or not less-than or equal). With these instructions, a QNaN source operand will generate an invalid-operation exception.

The invalid-operation exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

11.5.2.2 Denormal-Operand Exception (#D)

The processor signals the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand. The flag (DE) and mask (DM) bits for the denormal-operand exception are bits 1 and 8, respectively, in the MXCSR register.

The CVTPI2PD, CVTPD2PI, CVTTPD2PI, CVTDQ2PD, CVTPD2DQ, CVTTPD2DQ, CVTSI2SD, CVTSD2SI, CVTTSD2SI, CVTPI2PS, CVTPS2PI, CVTTPS2PI, CVTSS2SI, CVTTSS2SI, CVTSI2SS, CVTDQ2PS, CVTPS2DQ, and CVTTPS2DQ conversion instructions do not signal denormal exceptions. The RCPSS, RCPPS, RSQRTSS, and RSQRTPS instructions do not signal any kind of floating-point exception.

The denormals-are-zero flag (bit 6) of the MXCSR register provides an additional option for handling denormal-operand exceptions. When this flag is set, denormal source operands are automatically converted to zeros with the sign of the source operand (see Section 10.2.3.4, "Denormals-Are-Zeros"). The denormal operand exception is not affected by the flush-to-zero mode.

See Section 4.9.1.2, "Denormal Operand Exception (#D)," for more information about the denormal exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

11.5.2.3 Divide-By-Zero Exception (#Z)

The processor reports a divide-by-zero exception when a DIVPS, DIVSS, DIVPD or DIVSD instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) and mask (ZM) bits for the divide-by-zero exception are bits 2 and 9, respectively, in the MXCSR register.

See Section 4.9.1.3, "Divide-By-Zero Exception (#Z)," for more information about the divide-by-zero exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

The divide-by-zero exception is not affected by the flush-to-zero mode at a single-instruction boundary.

While DAZ does not affect the rules for signaling IEEE exceptions, operations on denormal inputs might have different results when DAZ=1. As a consequence, DAZ can have an effect on the floating-point exceptions - including the divide-by-zero exception - when observed for a given operation involving denormal inputs.

11.5.2.4 Numeric Overflow Exception (#O)

The processor reports a numeric overflow exception whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that fits in the destination operand. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTPD2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD and HSUBPS instructions. The flag (OE) and mask (OM) bits for the numeric overflow exception are bits 3 and 10, respectively, in the MXCSR register.

See Section 4.9.1.4, "Numeric Overflow Exception (#O)," for more information about the numeric-overflow exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

The numeric overflow exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

11.5.2.5 Numeric Underflow Exception (#U)

The processor reports a numeric underflow exception whenever the magnitude of the rounded result of an arithmetic instruction, with unbounded exponent, is less than the smallest possible normalized, finite value that will fit in the destination operand and the numeric-underflow exception is not masked. If the numeric underflow exception is masked, both underflow and the inexact-result condition must be detected before numeric underflow is reported. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTPD2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD, and HSUBPS instructions. The flag (UE) and mask (UM) bits for the numeric underflow exception are bits 4 and 11, respectively, in the MXCSR register.

The flush-to-zero flag (bit 15) of the MXCSR register provides an additional option for handling numeric underflow exceptions. When this flag is set and the numeric underflow exception is masked, tiny results are returned as a zero with the sign of the true result (see Section 10.2.3.3, "Flush-To-Zero").

Underflow will occur when a tiny non-zero result is detected (the result has to be also inexact if underflow exceptions are masked), as described in the IEEE Standard 754-2008. While DAZ does not affect the rules for signaling IEEE exceptions, operations on denormal inputs might have different results when DAZ=1. As a consequence, DAZ can have an effect on the floating-point exceptions - including the underflow exception - when observed for a given operation involving denormal inputs.

See Section 4.9.1.5, "Numeric Underflow Exception (#U)," for more information about the numeric underflow exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

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

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (normally acceptable) accuracy has been lost. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.

The flag (PE) and mask (PM) bits for the inexact-result exception are bits 2 and 12, respectively, in the MXCSR register.

See Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)," for more information about the inexact-result exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

In flush-to-zero mode, the inexact result exception is reported.

11.5.3 Generating SIMD Floating-Point Exceptions

When the processor executes a packed or scalar floating-point instruction, it looks for and reports on SIMD floating-point exception conditions using two sequential steps:

1. Looks for, reports on, and handles pre-computation exception conditions (invalid-operand, divide-by-zero, and denormal operand)
2. Looks for, reports on, and handles post-computation exception conditions (numeric overflow, numeric underflow, and inexact result)

If both pre- and post-computational exceptions are unmasked, it is possible for the processor to generate a SIMD floating-point exception (#XM) twice during the execution of an SSE, SSE2 or SSE3 instruction: once when it detects and handles a pre-computational exception and when it detects a post-computational exception.

11.5.3.1 Handling Masked Exceptions

If all exceptions are masked, the processor handles the exceptions it detects by placing the masked result (or results for packed operands) in a destination operand and continuing program execution. The masked result may be a rounded normalized value, signed infinity, a denormal finite number, zero, a QNaN floating-point indefinite, or

a QNaN depending on the exception condition detected. In most cases, the corresponding exception flag bit in MXCSR is also set. The one situation where an exception flag is not set is when an underflow condition is detected and it is not accompanied by an inexact result.

When operating on packed floating-point operands, the processor returns a masked result for each of the sub-operand computations and sets a separate set of internal exception flags for each computation. It then performs a logical-OR on the internal exception flag settings and sets the exception flags in the MXCSR register according to the results of OR operations.

For example, Figure 11-9 shows the results of an MULPS instruction. In the example, all SIMD floating-point exceptions are masked. Assume that a denormal exception condition is detected prior to the multiplication of sub-operands X0 and Y0, no exception condition is detected for the multiplication of X1 and Y1, a numeric overflow exception condition is detected for the multiplication of X2 and Y2, and another denormal exception is detected prior to the multiplication of sub-operands X3 and Y3. Because denormal exceptions are masked, the processor uses the denormal source values in the multiplications of (X0 and Y0) and of (X3 and Y3) passing the results of the multiplications through to the destination operand. With the denormal operand, the result of the X0 and Y0 computation is a normalized finite value, with no exceptions detected. However, the X3 and Y3 computation produces a tiny and inexact result. This causes the corresponding internal numeric underflow and inexact-result exception flags to be set.

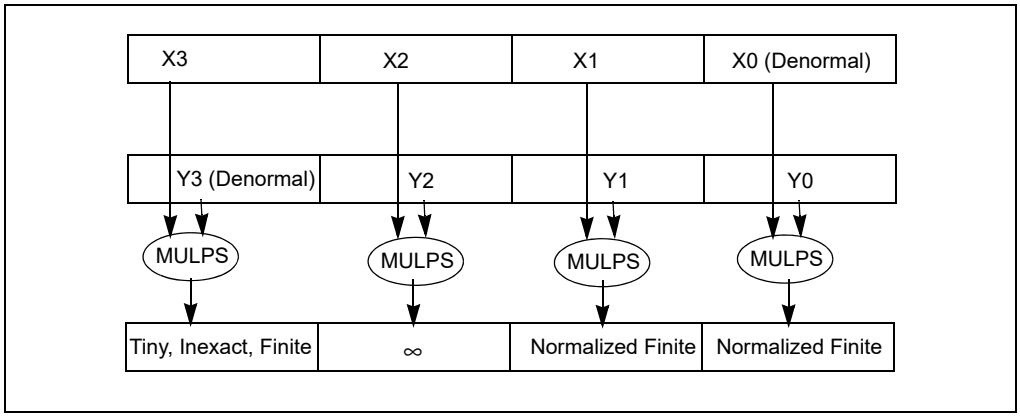


Figure 11-9. Example Masked Response for Packed Operations

For the multiplication of X2 and Y2, the processor stores the floating-point ∞ in the destination operand, and sets the corresponding internal sub-operand numeric overflow flag. The result of the X1 and Y1 multiplication is passed through to the destination operand, with no internal sub-operand exception flags being set. Following the computations, the individual sub-operand exceptions flags for denormal operand, numeric underflow, inexact result, and numeric overflow are OR'd and the corresponding flags are set in the MXCSR register.

The net result of this computation is that:

- Multiplication of X0 and Y0 produces a normalized finite result
- Multiplication of X1 and Y1 produces a normalized finite result
- Multiplication of X2 and Y2 produces a floating-point ∞ result
- Multiplication of X3 and Y3 produces a tiny, inexact, finite result
- Denormal operand, numeric underflow, numeric underflow, and inexact result flags are set in the MXCSR register

11.5.3.2 Handling Unmasked Exceptions

If all exceptions are unmasked, the processor:

1. First detects any pre-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and goes to step 2. If it does not detect any pre-computation exceptions, it goes to step 5.

2. Checks CR4.OSXMMEXCPT[bit 10]. If this flag is set, the processor goes to step 3; if the flag is clear, it generates an invalid-opcode exception (#UD) and makes an implicit call to the invalid-opcode exception handler.
3. Generates a SIMD floating-point exception (#XM) and makes an implicit call to the SIMD floating-point exception handler.
4. If the exception handler is able to fix the source operands that generated the pre-computation exceptions or mask the condition in such a way as to allow the processor to continue executing the instruction, the processor resumes instruction execution as described in step 5.
5. Upon returning from the exception handler (or if no pre-computation exceptions were detected), the processor checks for post-computation exceptions. If the processor detects any post-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and repeats steps 2, 3, and 4.
6. Upon returning from the exceptions handler in step 4 (or if no post-computation exceptions were detected), the processor completes the execution of the instruction.

The implication of this procedure is that for unmasked exceptions, the processor can generate a SIMD floating-point exception (#XM) twice: once if it detects pre-computation exception conditions and a second time if it detects post-computation exception conditions. For example, if SIMD floating-point exceptions are unmasked for the computation shown in Figure 11-9, the processor would generate one SIMD floating-point exception for denormal operand conditions and a second SIMD floating-point exception for overflow and underflow (no inexact result exception would be generated because the multiplications of X0 and Y0 and of X1 and Y1 are exact).

11.5.3.3 Handling Combinations of Masked and Unmasked Exceptions

In situations where both masked and unmasked exceptions are detected, the processor will set exception flags for the masked and the unmasked exceptions. However, it will not return masked results until after the processor has detected and handled unmasked post-computation exceptions and returned from the exception handler (as in step 6 above) to finish executing the instruction.

11.5.4 Handling SIMD Floating-Point Exceptions in Software

Section 4.9.3, "Typical Actions of a Floating-Point Exception Handler," shows actions that may be carried out by a SIMD floating-point exception handler. The SSE/SSE2/SSE3 state is saved with the FXSAVE instruction (see Section 11.6.5, "Saving and Restoring the SSE/SSE2 State").

11.5.5 Interaction of SIMD and x87 FPU Floating-Point Exceptions

SIMD floating-point exceptions are generated independently from x87 FPU floating-point exceptions. SIMD floating-point exceptions do not cause assertion of the FERR# pin (independent of the value of CR0.NE[bit 5]). They ignore the assertion and deassertion of the IGNNE# pin.

If applications use SSE/SSE2/SSE3 instructions along with x87 FPU instructions (in the same task or program), consider the following:

- SIMD floating-point exceptions are reported independently from the x87 FPU floating-point exceptions. SIMD and x87 FPU floating-point exceptions can be unmasked independently. Separate x87 FPU and SIMD floating-point exception handlers must be provided if the same exception is unmasked for x87 FPU and for SSE/SSE2/SSE3 operations.
- The rounding mode specified in the MXCSR register does not affect x87 FPU instructions. Likewise, the rounding mode specified in the x87 FPU control word does not affect the SSE/SSE2/SSE3 instructions. To use the same rounding mode, the rounding control bits in the MXCSR register and in the x87 FPU control word must be set explicitly to the same value.
- The flush-to-zero mode set in the MXCSR register for SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the flush-to-zero bit to 0.

- The denormals-are-zeros mode set in the MXCSR register for SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the denormals-are-zeros bit to 0.
- An application that expects to detect x87 FPU exceptions that occur during the execution of x87 FPU instructions will not be notified if exceptions occurs during the execution of corresponding SSE/SSE2/SSE3[1] instructions, unless the exception masks that are enabled in the x87 FPU control word have also been enabled in the MXCSR register and the application is capable of handling SIMD floating-point exceptions (#XM).
  - — Masked exceptions that occur during an SSE/SSE2/SSE3 library call cannot be detected by unmasking the exceptions after the call (in an attempt to generate the fault based on the fact that an exception flag is set). A SIMD floating-point exception flag that is set when the corresponding exception is unmasked will not generate a fault; only the next occurrence of that unmasked exception will generate a fault.
  - — An application which checks the x87 FPU status word to determine if any masked exception flags were set during an x87 FPU library call will also need to check the MXCSR register to detect a similar occurrence of a masked exception flag being set during an SSE/SSE2/SSE3 library call.

11.6 WRITING APPLICATIONS WITH SSE/SSE2 EXTENSIONS

The following sections give some guidelines for writing application programs and operating-system code that uses the SSE and SSE2 extensions. Because SSE and SSE2 extensions share the same state and perform companion operations, these guidelines apply to both sets of extensions.

Chapter 13 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, discusses the interface to the processor for context switching as well as other operating system considerations when writing code that uses SSE/SSE2/SSE3 extensions.

11.6.1 General Guidelines for Using SSE/SSE2 Extensions

The following guidelines describe how to take full advantage of the performance gains available with the SSE and SSE2 extensions:

- Ensure that the processor supports the SSE and SSE2 extensions.
- Ensure that your operating system supports the SSE and SSE2 extensions. (Operating system support for the SSE extensions implies support for SSE2 extension and vice versa.)
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Use the non-temporal store instructions offered with the SSE and SSE2 extensions.
- Employ the optimization and scheduling techniques described in the *Intel Pentium 4 Optimization Reference Manual* (see Section 1.4, "Related Literature," for the order number for this manual).

11.6.2 Checking for SSE/SSE2 Support

Before an application attempts to use the SSE and/or SSE2 extensions, it should check that they are present on the processor:

1. Check that the processor supports the CPUID instruction. Bit 21 of the EFLAGS register can be used to check processor's support the CPUID instruction.
2. Check that the processor supports the SSE and/or SSE2 extensions (true if CPUID.01H:EDX.SSE[bit 25] = 1 and/or CPUID.01H:EDX.SSE2[bit 26] = 1).

Operating system must provide system level support for handling SSE state, exceptions before an application can use the SSE and/or SSE2 extensions (see *Chapter 13* in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

1. SSE3 refers to ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD and HSUBPS; the only other SSE3 instruction that can raise floating-point exceptions is FISTTP: it can generate x87 FPU invalid operation and inexact result exceptions.

If the processor attempts to execute an unsupported SSE or SSE2 instruction, the processor will generate an invalid-opcode exception (#UD). If an operating system did not provide adequate system level support for SSE, executing an SSE or SSE2 instructions can also generate #UD.

11.6.3 Checking for the DAZ Flag in the MXCSR Register

The denormals-are-zero flag in the MXCSR register is available in most of the Pentium 4 processors and in the Intel Xeon processor, with the exception of some early steppings. To check for the presence of the DAZ flag in the MXCSR register, do the following:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See "FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for a description of the FXSAVE instruction and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   - If the value of the MXCSR_MASK field is 00000000H, the DAZ flag and denormals-are-zero mode are not supported.
   - If the value of the MXCSR_MASK field is non-zero and bit 6 is set, the DAZ flag and denormals-are-zero mode are supported.

If the DAZ flag is not supported, then it is a reserved bit and attempting to write a 1 to it will cause a general-protection exception (#GP). See Section 11.6.6, "Guidelines for Writing to the MXCSR Register," for general guidelines for preventing general-protection exceptions when writing to the MXCSR register.

11.6.4 Initialization of SSE/SSE2 Extensions

The SSE and SSE2 state is contained in the XMM and MXCSR registers. Upon a hardware reset of the processor, this state is initialized as follows (see Table 11-2):

- All SIMD floating-point exceptions are masked (bits 7 through 12 of the MXCSR register is set to 1).
- All SIMD floating-point exception flags are cleared (bits 0 through 5 of the MXCSR register is set to 0).
- The rounding control is set to round-nearest (bits 13 and 14 of the MXCSR register are set to 00B).
- The flush-to-zero mode is disabled (bit 15 of the MXCSR register is set to 0).
- The denormals-are-zeros mode is disabled (bit 6 of the MXCSR register is set to 0). If the denormals-are-zeros mode is not supported, this bit is reserved and will be set to 0 on initialization.
- Each of the XMM registers is cleared (set to all zeros).

Table 11-2. SSE and SSE2 State Following a Power-up/Reset or INIT

Registers	Power-Up or Reset	INIT
XMM0 through XMM7	+0.0	Unchanged
MXCSR	1F80H	Unchanged

If the processor is reset by asserting the INIT# pin, the SSE and SSE2 state is not changed.

11.6.5 Saving and Restoring the SSE/SSE2 State

The FXSAVE instruction saves the x87 FPU, MMX, SSE and SSE2 states (which includes the contents of eight XMM registers and the MXCSR registers) in a 512-byte block of memory. The FXRSTOR instruction restores the saved SSE and SSE2 state from memory. See the FXSAVE instruction in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for the layout of the 512-byte state block.

In addition to saving and restoring the SSE and SSE2 state, FXSAVE and FXRSTOR also save and restore the x87 FPU state (because MMX registers are aliased to the x87 FPU data registers this includes saving and restoring the MMX state). For greater code efficiency, it is suggested that FXSAVE and FXRSTOR be substituted for the FSAVE, FNSAVE and FRSTOR instructions in the following situations:

- When a context switch is being made in a multitasking environment
- During calls and returns from interrupt and exception handlers

In situations where the code is switching between x87 FPU and MMX technology computations (without a context switch or a call to an interrupt or exception), the FSAVE/FNSAVE and FRSTOR instructions are more efficient than the FXSAVE and FXRSTOR instructions.

11.6.6 Guidelines for Writing to the MXCSR Register

The MXCSR has several reserved bits, and attempting to write a 1 to any of these bits will cause a general-protection exception (#GP) to be generated. To allow software to identify these reserved bits, the MXCSR_MASK value is provided. Software can determine this mask value as follows:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See "FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for a description of FXSAVE and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   - If the value of the MXCSR_MASK field is 00000000H, then the MXCSR_MASK value is the default value of 0000FFBFH. Note that this value indicates that bit 6 of the MXCSR register is reserved; this setting indicates that the denormals-are-zero mode is not supported on the processor.
   - If the value of the MXCSR_MASK field is non-zero, the MXCSR_MASK value should be used as the MXCSR_MASK.

All bits set to 0 in the MXCSR_MASK value indicate reserved bits in the MXCSR register. Thus, if the MXCSR_MASK value is AND'd with a value to be written into the MXCSR register, the resulting value will be assured of having all its reserved bits set to 0, preventing the possibility of a general-protection exception being generated when the value is written to the MXCSR register.

For example, the default MXCSR_MASK value when 00000000H is returned in the FXSAVE image is 0000FFBFH. If software AND's a value to be written to MXCSR register with 0000FFBFH, bit 6 of the result (the DAZ flag) will be ensured of being set to 0, which is the required setting to prevent general-protection exceptions on processors that do not support the denormals-are-zero mode.

To prevent general-protection exceptions, the MXCSR_MASK value should be AND'd with the value to be written into the MXCSR register in the following situations:

- Operating system routines that receive a parameter from an application program and then write that value to the MXCSR register (either with an FXRSTOR or LDMXCSR instruction)
- Any application program that writes to the MXCSR register and that needs to run robustly on several different IA-32 processors

Note that all bits in the MXCSR_MASK value that are set to 1 indicate features that are supported by the MXCSR register; they can be treated as feature flags for identifying processor capabilities.

11.6.7 Interaction of SSE/SSE2 Instructions with x87 FPU and MMX Instructions

The XMM registers and the x87 FPU and MMX registers represent separate execution environments, which has certain ramifications when executing SSE, SSE2, MMX, and x87 FPU instructions in the same code module or when mixing code modules that contain these instructions:

- Those SSE and SSE2 instructions that operate only on XMM registers (such as the packed and scalar floating-point instructions and the 128-bit SIMD integer instructions) in the same instruction stream with 64-bit SIMD integer or x87 FPU instructions without any restrictions. For example, an application can perform the majority of its floating-point computations in the XMM registers, using the packed and scalar floating-point instructions, and at the same time use the x87 FPU to perform trigonometric and other transcendental computations. Likewise, an application can perform packed 64-bit and 128-bit SIMD integer operations together without restrictions.
- Those SSE and SSE2 instructions that operate on MMX registers (such as the CVTPS2PI, CVTTPS2PI, CVTPI2PS, CVTPD2PI, CVTTPD2PI, CVTPI2PD, MOVDQ2Q, MOVQ2DQ, PADDQ, and PSUBQ instructions) can also be executed in the same instruction stream as 64-bit SIMD integer or x87 FPU instructions, however, here they are subject to the restrictions on the simultaneous use of MMX technology and x87 FPU instructions, which include:
  - Transition from x87 FPU to MMX technology instructions or to SSE or SSE2 instructions that operate on MMX registers should be preceded by saving the state of the x87 FPU.
  - Transition from MMX technology instructions or from SSE or SSE2 instructions that operate on MMX registers to x87 FPU instructions should be preceded by execution of the EMMS instruction.

11.6.8 Compatibility of SIMD and x87 FPU Floating-Point Data Types

SSE and SSE2 extensions operate on the same single-precision and double-precision floating-point data types that the x87 FPU operates on. However, when operating on these data types, the SSE and SSE2 extensions operate on them in their native format (single-precision or double-precision), in contrast to the x87 FPU which extends them to double extended-precision floating-point format to perform computations and then rounds the result back to a single-precision or double-precision format before writing results to memory. Because the x87 FPU operates on a higher precision format and then rounds the result to a lower precision format, it may return a slightly different result when performing the same operation on the same single-precision or double-precision floating-point values than is returned by the SSE and SSE2 extensions. The difference occurs only in the least-significant bits of the significand.

11.6.9 Mixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data

SSE and SSE2 extensions define typed operations on packed and scalar floating-point data types and on 128-bit SIMD integer data types, but IA-32 processors do not enforce this typing at the architectural level. They only enforce it at the microarchitectural level. Therefore, when a Pentium 4 or Intel Xeon processor loads a packed or scalar floating-point operand or a 128-bit packed integer operand from memory into an XMM register, it does not check that the actual data being loaded matches the data type specified in the instruction. Likewise, when the processor performs an arithmetic operation on the data in an XMM register, it does not check that the data being operated on matches the data type specified in the instruction.

As a general rule, because data typing of SIMD floating-point and integer data types is not enforced at the architectural level, it is the responsibility of the programmer, assembler, or compiler to insure that code enforces data typing. Failure to enforce correct data typing can lead to computations that return unexpected results.

For example, in the following code sample, two packed single-precision floating-point operands are moved from memory into XMM registers (using MOVAPS instructions); then a double-precision packed add operation (using the ADDPD instruction) is performed on the operands:

```
movaps      xmm0, [eax]   ; EAX register contains pointer to packed
                          ; single-precision floating-point operand
movaps      xmm1, [ebx]
addpd       xmm0, xmm1
```

Pentium 4 and Intel Xeon processors execute these instructions without generating an invalid-operand exception (#UD) and will produce the expected results in register XMM0 (that is, the high and low 64-bits of each register will be treated as a double-precision floating-point value and the processor will operate on them accordingly). Because the data types operated on and the data type expected by the ADDPD instruction were inconsistent, the instruction

may result in a SIMD floating-point exception (such as numeric overflow [#O] or invalid operation [#I]) being generated, but the actual source of the problem (inconsistent data types) is not detected.

The ability to operate on an operand that contains a data type that is inconsistent with the typing of the instruction being executed, permits some valid operations to be performed. For example, the following instructions load a packed double-precision floating-point operand from memory to register XMM0, and a mask to register XMM1; then they use XORPD to toggle the sign bits of the two packed values in register XMM0.

```
movapd      xmm0, [eax]   ; EAX register contains pointer to packed
                          ; double-precision floating-point operand
movaps      xmm1, [ebx]    ; EBX register contains pointer to packed
                           ; double-precision floating-point mask
xorpd       xmm0, xmm1    ; XOR operation toggles sign bits using
                          ; the mask in xmm1
```

In this example: XORPS or PXOR can be used in place of XORPD and yield the same correct result. However, because of the type mismatch between the operand data type and the instruction data type, a latency penalty will be incurred due to implementations of the instructions at the microarchitecture level.

Latency penalties can also be incurred by using move instructions of the wrong type. For example, MOVAPS and MOVAPD can both be used to move a packed single-precision operand from memory to an XMM register. However, if MOVAPD is used, a latency penalty will be incurred when a correctly typed instruction attempts to use the data in the register.

Note that these latency penalties are not incurred when moving data from XMM registers to memory.

11.6.10 Interfacing with SSE/SSE2 Procedures and Functions

SSE and SSE2 extensions allow direct access to XMM registers. This means that all existing interface conventions between procedures and functions that apply to the use of the general-purpose registers (EAX, EBX, etc.) also apply to XMM register usage.

11.6.10.1 Passing Parameters in XMM Registers

The state of XMM registers is preserved across procedure (or function) boundaries. Parameters can be passed from one procedure to another using XMM registers.

11.6.10.2 Saving XMM Register State on a Procedure or Function Call

The state of XMM registers can be saved in two ways: using an FXSAVE instruction or a move instruction. FXSAVE saves the state of all XMM registers (along with the state of MXCSR and the x87 FPU registers). This instruction is typically used for major changes in the context of the execution environment, such as a task switch. FXRSTOR restores the XMM, MXCSR, and x87 FPU registers stored with FXSAVE.

In cases where only XMM registers must be saved, or where selected XMM registers need to be saved, move instructions (MOVAPS, MOVUPS, MOVSS, MOVAPD, MOVUPD, MOVSD, MOVDQA, and MOVDQU) can be used. These instructions can also be used to restore the contents of XMM registers. To avoid performance degradation when saving XMM registers to memory or when loading XMM registers from memory, be sure to use the appropriately typed move instructions.

The move instructions can also be used to save the contents of XMM registers on the stack. Here, the stack pointer (in the ESP register) can be used as the memory address to the next available byte in the stack. Note that the stack pointer is not automatically incremented when using a move instruction (as it is with PUSH).

A move-instruction procedure that saves the contents of an XMM register to the stack is responsible for decrementing the value in the ESP register by 16. Likewise, a move-instruction procedure that loads an XMM register from the stack needs also to increment the ESP register by 16. To avoid performance degradation when moving the contents of XMM registers, use the appropriately typed move instructions.

Use the LDMXCSR and STMXCSR instructions to save and restore, respectively, the contents of the MXCSR register on a procedure call and return.

11.6.10.3 Caller-Save Recommendation for Procedure and Function Calls

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

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

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

11.6.11 Updating Existing MMX Technology Routines Using 128-Bit SIMD Integer Instructions

SSE2 extensions extend all 64-bit MMX SIMD integer instructions to operate on 128-bit SIMD integers using XMM registers. The extended 128-bit SIMD integer instructions operate like the 64-bit SIMD integer instructions; this simplifies the porting of MMX technology applications. However, there are considerations:

- To take advantage of wider 128-bit SIMD integer instructions, MMX technology code must be recompiled to reference the XMM registers instead of MMX registers.
- Computation instructions that reference memory operands that are not aligned on 16-byte boundaries should be replaced with an unaligned 128-bit load (MOVUDQ instruction) followed by a version of the same computation operation that uses register instead of memory operands. Use of 128-bit packed integer computation instructions with memory operands that are not 16-byte aligned results in a general protection exception (#GP).
- Extension of the PSHUFW instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: PSHUFHW, PSHUFLW, and PSHUFD.
- Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) can be extended to 128 bits in either of two ways:
  - Use of PSRLQ and PSLLQ, along with masking logic operations.
  - Rewriting the code sequence to use PSRLDQ and PSLLDQ (shift double quadword operand by bytes)
- Loop counters need to be updated, since each 128-bit SIMD integer instruction operates on twice the amount of data as its 64-bit SIMD integer counterpart.

11.6.12 Branching on Arithmetic Operations

There are no condition codes in SSE or SSE2 states. A packed-data comparison instruction generates a mask which can then be transferred to an integer register. The following code sequence provides an example of how to perform a conditional branch, based on the result of an SSE2 arithmetic operation.

```
cmppd       XMM0, XMM1          ; generates a mask in XMM0
movmskpd    EAX, XMM0           ; moves a 2 bit mask to eax
test        EAX, 0              ; compare with desired result
jne         BRANCH TARGET
```

The COMISD and UCOMISD instructions update the EFLAGS as the result of a scalar comparison. A conditional branch can then be scheduled immediately following COMISD/UCOMISD.

11.6.13 Cacheability Hint Instructions

SSE and SSE2 cacheability control instructions enable the programmer to control prefetching, caching, loading and storing of data. When correctly used, these instructions improve application performance.

To make efficient use of the processor's super-scalar microarchitecture, a program needs to provide a steady stream of data to the executing program to avoid stalling the processor. PREFETCH*h* instructions minimize the latency of data accesses in performance-critical sections of application code by allowing data to be fetched into the processor cache hierarchy in advance of actual usage.

PREFETCH*h* instructions do not change the user-visible semantics of a program, although they may affect performance. The operation of these instructions is implementation-dependent. Programmers may need to tune code for each IA-32 processor implementation. Excessive usage of PREFETCH*h* instructions may waste memory bandwidth and reduce performance. For more detailed information on the use of prefetch hints, refer to Chapter 7, "Optimizing Cache Usage,", in the *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

The non-temporal store instructions (MOVNTI, MOVNTPD, MOVNTPS, MOVNTDQ, MOVNTQ, MASKMOVQ, and MASKMOVDQU) minimize cache pollution when writing non-temporal data to memory (see Section 10.4.6.1, "Cacheability Control Instructions" and Section 10.4.6.2, "Caching of Temporal vs. Non-Temporal Data"). They prevent non-temporal data from being written into processor caches on a store operation.

Besides reducing cache pollution, the use of weakly-ordered memory types can be important under certain data sharing relationships, such as a producer-consumer relationship. The use of weakly ordered memory can make the assembling of data more efficient; but care must be taken to ensure that the consumer obtains the data that the producer intended. Some common usage models that may be affected in this way by weakly-ordered stores are:

- Library functions that use weakly ordered memory to write results
- Compiler-generated code that writes weakly-ordered results
- Hand-crafted code

The degree to which a consumer of data knows that the data is weakly ordered can vary for these cases. As a result, the SFENCE or MFENCE instruction should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume the data. SFENCE and MFENCE provide a performance-efficient way to ensure ordering by guaranteeing that every store instruction that precedes SFENCE/MFENCE in program order is globally visible before a store instruction that follows the fence.

11.6.14 Effect of Instruction Prefixes on the SSE/SSE2 Instructions

Table 11-3 describes the effects of instruction prefixes on SSE and SSE2 instructions. (Table 11-3 also applies to SIMD integer and SIMD floating-point instructions in SSE3.) Unpredictable behavior can range from prefixes being treated as a reserved operation on one generation of IA-32 processors to generating an invalid opcode exception on another generation of processors.

See also "Instruction Prefixes" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for complete description of instruction prefixes.

NOTE

Some SSE/SSE2/SSE3 instructions have two-byte opcodes that are either 2 bytes or 3 bytes in length. Two-byte opcodes that are 3 bytes in length consist of: a mandatory prefix (F2H, F3H, or 66H), 0FH, and an opcode byte. See Table 11-3.

Table 11-3. Effect of Prefixes on SSE, SSE2, and SSE3 Instructions

Prefix Type	Effect on SSE, SSE2 and SSE3 Instructions
Address Size Prefix (67H)	Affects instructions with a memory operand.
	Reserved for instructions without a memory operand and may result in unpredictable behavior.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects instructions with a memory operand.
	Reserved for instructions without a memory operand and may result in unpredictable behavior.
Repeat Prefixes (F2H and F3H)	Reserved and may result in unpredictable behavior.
Lock Prefix (F0H)	Reserved; generates invalid opcode exception (#UD).
Branch Hint Prefixes(E2H and E3H)	Reserved and may result in unpredictable behavior.

6. Updates to Chapter 19, Volume 1

Change bars show changes to Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Change to this chapter: Defined the "NPX" acronym: Numeric Processor Extensions.

CHAPTER 19
PROCESSOR IDENTIFICATION AND FEATURE DETERMINATION

When writing software intended to run on IA-32 processors, it is necessary to identify the type of processor present in a system and the processor features that are available to an application.

19.1 USING THE CPUID INSTRUCTION

Use the CPUID instruction for processor identification in the Pentium M processor family, Pentium 4 processor family, Intel Xeon processor family, P6 family, Pentium processor, and later Intel486 processors. This instruction returns the family, model and (for some processors) a brand string for the processor that executes the instruction. It also indicates the features that are present in the processor and gives information about the processor's caches and TLB.

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. The CPUID instruction will cause the invalid opcode exception (#UD) if executed on a processor that does not support it.

To obtain processor identification information, a source operand value is placed in the EAX register to select the type of information to be returned. When the CPUID instruction is executed, selected information is returned in the EAX, EBX, ECX, and EDX registers. For a complete description of the CPUID instruction, tables indicating values returned, and example code, see *CPUID—CPU Identification* in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

19.1.1 Notes on Where to Start

The following guidelines are among the most important, and should always be followed when using the CPUID instruction to determine available features:

- Always begin by testing for the "GenuineIntel," message in the EBX, EDX, and ECX registers when the CPUID instruction is executed with EAX equal to 0. If the processor is not genuine Intel, the feature identification flags may have different meanings than are described in Intel documentation.
- Test feature identification flags individually and do not make assumptions about undefined bits.

19.1.2 Identification of Earlier IA-32 Processors

The CPUID instruction is not available in earlier IA-32 processors up through the earlier Intel486 processors. For these processors, several other architectural features can be exploited to identify the processor.

The settings of bits 12 and 13 (IOPL), 14 (NT), and 15 (reserved) in the EFLAGS register are different for Intel's 32-bit processors than for the Intel 8086 and Intel 286 processors. By examining the settings of these bits (with the PUSHF/PUSHFD and POPF/POPFD instructions), an application program can determine whether the processor is an 8086, Intel 286, or one of the Intel 32-bit processors:

- 8086 processor — Bits 12 through 15 of the EFLAGS register are always set.
- Intel 286 processor — Bits 12 through 15 are always clear in real-address mode.
- 32-bit processors — In real-address mode, bit 15 is always clear and bits 12 through 14 have the last value loaded into them. In protected mode, bit 15 is always clear, bit 14 has the last value loaded into it, and the IOPL bits depend on the current privilege level (CPL). The IOPL field can be changed only if the CPL is 0.

Other EFLAGS register bits that can be used to differentiate between the 32-bit processors:

- Bit 18 (AC) — Implemented only on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The inability to set or clear this bit distinguishes an Intel386 processor from the later IA-32 processors.
- Bit 21 (ID) — Determines if the processor is able to execute the CPUID instruction. The ability to set and clear this bit indicates that it is a Pentium 4, Intel Xeon, P6 family, Pentium, or later-version Intel486 processor.

To determine whether an x87 FPU or Numeric Processor Extension (NPX) is present in a system, applications can write to the x87 FPU status and control registers using the FNINIT instruction and then verify that the correct values are read back using the FNSTENV instruction.

After determining that an x87 FPU or NPX is present, its type can then be determined. In most cases, the processor type will determine the type of FPU or NPX; however, an Intel386 processor is compatible with either an Intel 287 or Intel 387 math coprocessor.

The method the coprocessor uses to represent ∞ (after the execution of the FINIT, FNINIT, or RESET instruction) indicates which coprocessor is present. The Intel 287 math coprocessor uses the same bit representation for +∞ and –∞; whereas, the Intel 387 math coprocessor uses different representations for +∞ and –∞.

7. Updates to Appendix D, Volume 1

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

Change to this chapter: Defined the "NPX" acronym: Numeric Processor Extensions.

APPENDIX D
GUIDELINES FOR WRITING X87 FPU EXCEPTION HANDLERS

As described in Chapter 8, "Programming with the x87 FPU," the IA-32 Architecture supports two mechanisms for accessing exception handlers to handle unmasked x87 FPU exceptions: native mode and MS-DOS compatibility mode. The primary purpose of this appendix is to provide detailed information to help software engineers design and write x87 FPU exception-handling facilities to run on PC systems that use the MS-DOS compatibility mode[1] for handling x87 FPU exceptions. Some of the information in this appendix will also be of interest to engineers who are writing native-mode x87 FPU exception handlers. The information provided is as follows:

- Discussion of the origin of the MS-DOS x87 FPU exception handling mechanism and its relationship to the x87 FPU's native exception handling mechanism.
- Description of the IA-32 flags and processor pins that control the MS-DOS x87 FPU exception handling mechanism.
- Description of the external hardware typically required to support MS-DOS exception handling mechanism.
- Description of the x87 FPU's exception handling mechanism and the typical protocol for x87 FPU exception handlers.
- Code examples that demonstrate various levels of x87 FPU exception handlers.
- Discussion of x87 FPU considerations in multitasking environments.
- Discussion of native mode x87 FPU exception handling.

The information given is oriented toward the most recent generations of IA-32 processors, starting with the Intel486. It is intended to augment the reference information given in Chapter 8, "Programming with the x87 FPU."

A more extensive version of this appendix is available in the application note AP-578, *Software and Hardware Considerations for x87 FPU Exception Handlers for Intel Architecture Processors* (Order Number 243291), which is available from Intel.

D.1 MS-DOS COMPATIBILITY SUB-MODE FOR HANDLING X87 FPU EXCEPTIONS

The first generations of IA-32 processors (starting with the Intel 8086 and 8088 processors and going through the Intel 286 and Intel386 processors) did not have an on-chip floating-point unit. Instead, floating-point capability was provided on a separate numeric coprocessor chip. The first of these numeric coprocessors was the Intel 8087, which was followed by the Intel 287 and Intel 387 numeric coprocessors.

To allow the 8087 to signal floating-point exceptions to its companion 8086 or 8088, the 8087 has an output pin, INT, which it asserts when an unmasked floating-point exception occurs. The designers of the 8087 recommended that the output from this pin be routed through a programmable interrupt controller (PIC) such as the Intel 8259A to the INTR pin of the 8086 or 8088. The handler for the resulting interrupt could then be used to access the floating-point exception handler.

However, the original IBM* PC design and MS-DOS operating system used a different mechanism for handling the INT output from the 8087. It connected the INT pin directly to the NMI input pin of the 8086 or 8088. The NMI interrupt handler then had to determine if the interrupt was caused by a floating-point exception or another NMI event. This mechanism is the origin of what is now called the "MS-DOS compatibility mode." The decision to use this latter floating-point exception handling mechanism came about because when the IBM PC was first designed, the 8087 was not available. When the 8087 did become available, other functions had already been assigned to the eight inputs to the PIC. One of these functions was a BIOS video interrupt, which was assigned vector 16 for the 8086 and 8088.

1 Microsoft Windows* 95 and Windows 3.1 (and earlier versions) operating systems use almost the same x87 FPU exception handling interface as MS-DOS. The recommendations in this appendix for a MS-DOS compatible exception handler thus apply to all three operating systems.

The Intel 286 processor created the "native mode" for handling floating-point exceptions by providing a dedicated input pin (ERROR#) for receiving floating-point exception signals and a dedicated interrupt vector, 16. Interrupt 16 was used to signal floating-point errors (also called math faults). It was intended that the ERROR# pin on the Intel 286 be connected to a corresponding ERROR# pin on the Intel 287 numeric coprocessor. When the Intel 287 signals a floating-point exception using this mechanism, the Intel 286 generates an interrupt 16, to invoke the floating-point exception handler.

To maintain compatibility with existing PC software, the native floating-point exception handling mode of the Intel 286 and 287 was not used in the IBM PC AT system design. Instead, the ERROR# pin on the Intel 286 was tied permanently high, and the ERROR# pin from the Intel 287 was routed to a second (cascaded) PIC. The resulting output of this PIC was routed through an exception handler and eventually caused an interrupt 2 (NMI interrupt). Here the NMI interrupt was shared with IBM PC AT's new parity checking feature. Interrupt 16 remained assigned to the BIOS video interrupt handler. The external hardware for the MS-DOS compatibility mode must prevent the Intel 286 processor from executing past the next x87 FPU instruction when an unmasked exception has been generated. To do this, it asserts the BUSY# signal into the Intel 286 when the ERROR# signal is asserted by the Intel 287.

The Intel386 processor and its companion Intel 387 numeric coprocessor provided the same hardware mechanism for signaling and handling floating-point exceptions as the Intel 286 and 287 processors. And again, to maintain compatibility with existing MS-DOS software, basically the same MS-DOS compatibility floating-point exception handling mechanism that was used in the IBM PC AT was used in PCs based on the Intel386 processor.

D.2 IMPLEMENTATION OF THE MS-DOS* COMPATIBILITY SUB-MODE IN THE INTEL486™, PENTIUM®, AND P6 PROCESSOR FAMILY, AND PENTIUM® 4 PROCESSORS

Beginning with the Intel486™ processor, the IA-32 architecture provided a dedicated mechanism for enabling the MS-DOS compatibility mode for x87 FPU exceptions and for generating external x87 FPU-exception signals while operating in this mode. The following sections describe the implementation of the MS-DOS compatibility mode in the Intel486 and Pentium processors and in the P6 family and Pentium 4 processors. Also described is the recommended external hardware to support this mode of operation.

D.2.1 MS-DOS* Compatibility Sub-mode in the Intel486™ and Pentium® Processors

In the Intel486 processor, several things were done to enhance and speed up the numeric coprocessor, now called the floating-point unit (x87 FPU). The most important enhancement was that the x87 FPU was included in the same chip as the processor, for increased speed in x87 FPU computations and reduced latency for x87 FPU exception handling. Also, for the first time, the MS-DOS compatibility mode was built into the chip design, with the addition of the NE bit in control register CR0 and the addition of the FERR# (Floating-point ERRor) and IGNNE# (IGNore Numeric Error) pins.

The NE bit selects the native x87 FPU exception handling mode (NE = 1) or the MS-DOS compatibility mode (NE = 0). When native mode is selected, all signaling of floating-point exceptions is handled internally in the Intel486 chip, resulting in the generation of an interrupt 16.

When MS-DOS compatibility mode is selected, the FERRR# and IGNNE# pins are used to signal floating-point exceptions. The FERR# output pin, which replaces the ERROR# pin from the previous generations of IA-32 numeric coprocessors, is connected to a PIC. A new input signal, IGNNE#, is provided to allow the x87 FPU exception handler to execute x87 FPU instructions, if desired, without first clearing the error condition and without triggering the interrupt a second time. This IGNNE# feature is needed to replicate the capability that was provided on MS-DOS compatible Intel 286 and Intel 287 and Intel386 and Intel 387 systems by turning off the BUSY# signal, when inside the x87 FPU exception handler, before clearing the error condition.

Note that Intel, in order to provide Intel486 processors for market segments that had no need for an x87 FPU, created the "SX" versions. These Intel486 SX processors did not contain the floating-point unit. Intel also produced Intel 487 SX processors for end users who later decided to upgrade to a system with an x87 FPU. These Intel 487 SX processors are similar to standard Intel486 processors with a working x87 FPU on board.

Thus, the external circuitry necessary to support the MS-DOS compatibility mode for Intel 487 SX processors is the same as for standard Intel486 DX processors.

The Pentium, P6 family, and Pentium 4 processors offer the same mechanism (the NE bit and the FERR# and IGNNE# pins) as the Intel486 processors for generating x87 FPU exceptions in MS-DOS compatibility mode. The actions of these mechanisms are slightly different and more straightforward for the P6 family and Pentium 4 processors, as described in Section D.2.2, "MS-DOS* Compatibility Sub-mode in the P6 Family and Pentium® 4 Processors."

For Pentium, P6 family, and Pentium 4 processors, it is important to note that the special DP (Dual Processing) mode for Pentium processors and also the more general Intel MultiProcessor Specification for systems with multiple Pentium, P6 family, or Pentium 4 processors support x87 FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility x87 FPU mode for systems using more than one processor.

D.2.1.1 Basic Rules: When FERR# Is Generated

When MS-DOS compatibility mode is enabled for the Intel486 or Pentium processors (NE bit is set to 0) and the IGNNE# input pin is de-asserted, the FERR# signal is generated as follows:

1. When an x87 FPU instruction causes an unmasked x87 FPU exception, the processor (in most cases) uses a "deferred" method of reporting the error. This means that the processor does not respond immediately, but rather freezes just before executing the next WAIT or x87 FPU instruction (except for "no-wait" instructions, which the x87 FPU executes regardless of an error condition).
2. When the processor freezes, it also asserts the FERR# output.
3. The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# assertion.
4. In MS-DOS compatibility systems, FERR# is fed to the IRQ13 input in the cascaded PIC. The PIC generates interrupt 75H, which then branches to interrupt 2, as described earlier in this appendix for systems using the Intel 286 and Intel 287 or Intel386 and Intel 387 processors.

The deferred method of error reporting is used for all exceptions caused by the basic arithmetic instructions (including FADD, FSUB, FMUL, FDIV, FSQRT, FCOM and FUCOM), for precision exceptions caused by all types of x87 FPU instructions, and for numeric underflow and overflow exceptions caused by all types of x87 FPU instructions except stores to memory.

Some x87 FPU instructions with some x87 FPU exceptions use an "immediate" method of reporting errors. Here, the FERR# is asserted immediately, at the time that the exception occurs. The immediate method of error reporting is used for x87 FPU stack fault, invalid operation and denormal exceptions caused by all transcendental instructions, FSCALE, FXTRACT, FPREM and others, and all exceptions (except precision) when caused by x87 FPU store instructions. Like deferred error reporting, immediate error reporting will cause the processor to freeze just before executing the next WAIT or x87 FPU instruction if the error condition has not been cleared by that time.

Note that in general, whether deferred or immediate error reporting is used for an x87 FPU exception depends both on which exception occurred and which instruction caused that exception. A complete specification of these cases, which applies to both the Pentium and the Intel486 processors, is given in Section 5.1.21 in the *Pentium Processor Family Developer's Manual: Volume 1*.

If NE = 0 but the IGNNE# input is active while an unmasked x87 FPU exception is in effect, the processor disregards the exception, does not assert FERR#, and continues. If IGNNE# is then de-asserted and the x87 FPU exception has not been cleared, the processor will respond as described above. (That is, an immediate exception case will assert FERR# immediately. A deferred exception case will assert FERR# and freeze just before the next x87 FPU or WAIT instruction.) The assertion of IGNNE# is intended for use only inside the x87 FPU exception handler, where it is needed if one wants to execute non-control x87 FPU instructions for diagnosis, before clearing the exception condition. When IGNNE# is asserted inside the exception handler, a preceding x87 FPU exception has already caused FERR# to be asserted, and the external interrupt hardware has responded, but IGNNE# assertion still prevents the freeze at x87 FPU instructions. Note that if IGNNE# is left active outside of the x87 FPU exception handler, additional x87 FPU instructions may be executed after a given instruction has caused an x87 FPU exception. In this case, if the x87 FPU exception handler ever did get invoked, it could not determine which instruction caused the exception.

To properly manage the interface between the processor's FERR# output, its IGNNE# input, and the IRQ13 input of the PIC, additional external hardware is needed. A recommended configuration is described in the following section.

D.2.1.2 Recommended External Hardware to Support the MS-DOS* Compatibility Sub-mode

Figure D-1 provides an external circuit that will assure proper handling of FERR# and IGNNE# when an x87 FPU exception occurs. In particular, it assures that IGNNE# will be active only inside the x87 FPU exception handler without depending on the order of actions by the exception handler. Some hardware implementations have been less robust because they have depended on the exception handler to clear the x87 FPU exception interrupt request to the PIC (FP_IRQ signal) **before** the handler causes FERR# to be de-asserted by clearing the exception from the x87 FPU itself. Figure D-2 shows the details of how IGNNE# will behave when the circuit in Figure D-1 is implemented. The temporal regions within the x87 FPU exception handler activity are described as follows:

1. The FERR# signal is activated by an x87 FPU exception and sends an interrupt request through the PIC to the processor's INTR pin.
2. During the x87 FPU interrupt service routine (exception handler) the processor will need to clear the interrupt request latch (Flip Flop #1). It may also want to execute non-control x87 FPU instructions before the exception is cleared from the x87 FPU. For this purpose the IGNNE# must be driven low. Typically in the PC environment an I/O access to Port 0F0H clears the external x87 FPU exception interrupt request (FP_IRQ). In the recommended circuit, this access also is used to activate IGNNE#. With IGNNE# active, the x87 FPU exception handler may execute any x87 FPU instruction without being blocked by an active x87 FPU exception.
3. Clearing the exception within the x87 FPU will cause the FERR# signal to be deactivated and then there is no further need for IGNNE# to be active. In the recommended circuit, the deactivation of FERR# is used to deactivate IGNNE#. If another circuit is used, the software and circuit together must assure that IGNNE# is deactivated no later than the exit from the x87 FPU exception handler.

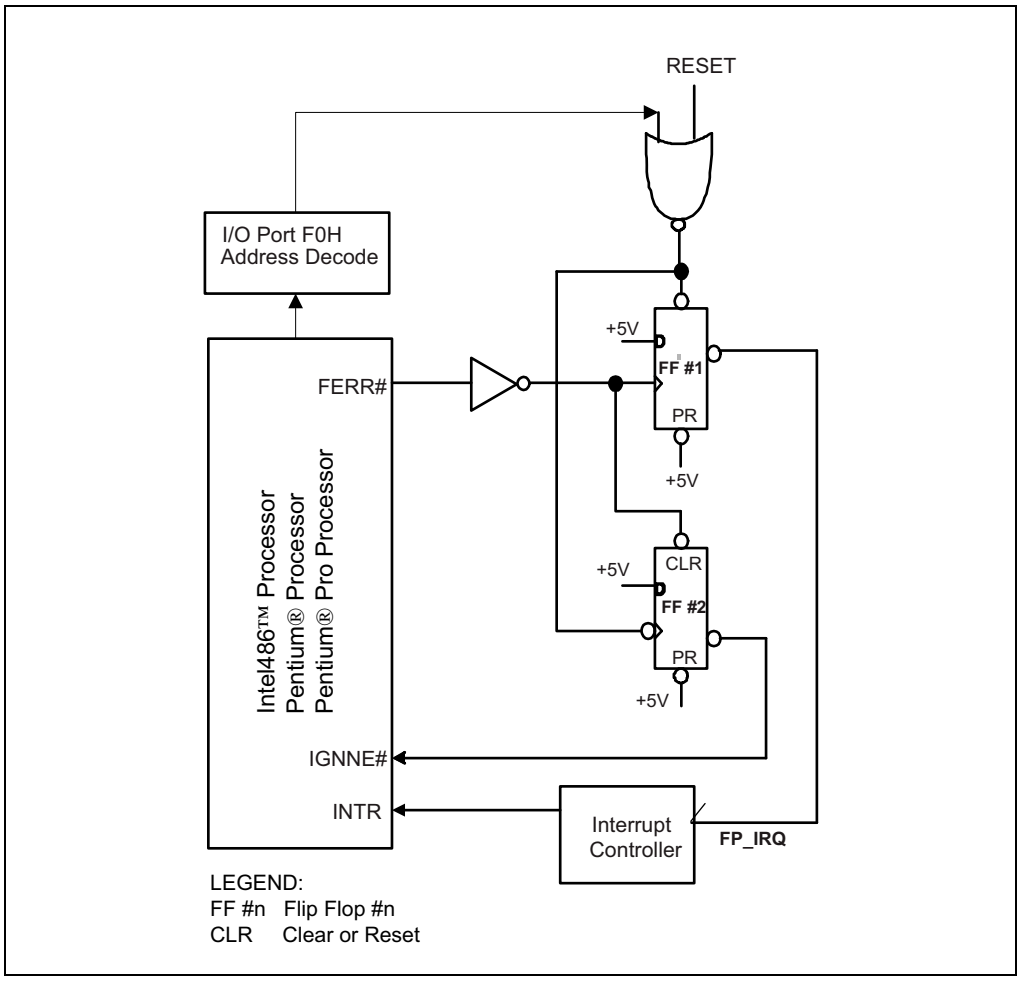


Figure D-1. Recommended Circuit for MS-DOS Compatibility x87 FPU Exception Handling

In the circuit in Figure D-1, when the x87 FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. So the handler can activate IGNNE#, if needed, by doing this 0F0H access before clearing the x87 FPU exception condition (which de-asserts FERR#).

However, the circuit does not depend on the order of actions by the x87 FPU exception handler to guarantee the correct hardware state upon exit from the handler. Flip Flop #2, which drives IGNNE# to the processor, has its CLEAR input attached to the inverted FERR#. This ensures that IGNNE# can never be active when FERR# is inactive. So if the handler clears the x87 FPU exception condition **before** the 0F0H access, IGNNE# does not get activated and left on after exit from the handler.

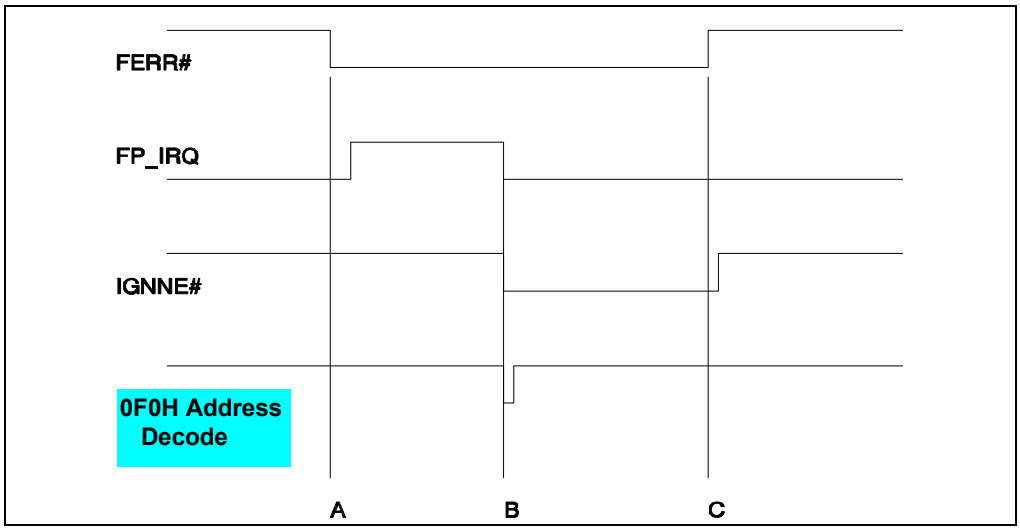


Figure D-2. Behavior of Signals During x87 FPU Exception Handling

D.2.1.3 No-Wait x87 FPU Instructions Can Get x87 FPU Interrupt in Window

The Pentium and Intel486 processors implement the "no-wait" floating-point instructions (FNINIT, FNCLEX, FNSTENV, FNSAVE, FNSTSW, FNSTCW, FNENI, FNDISI or FNSETPM) in the MS-DOS compatibility mode in the following manner. (See Section 8.3.11, "x87 FPU Control Instructions," and Section 8.3.12, "Waiting vs. Non-waiting Instructions," for a discussion of the no-wait instructions.)

If an unmasked numeric exception is pending from a preceding x87 FPU instruction, a member of the no-wait class of instructions will, at the beginning of its execution, assert the FERR# pin in response to that exception just like other x87 FPU instructions, but then, unlike the other x87 FPU instructions, FERR# will be de-asserted. This de-assertion was implemented to allow the no-wait class of instructions to proceed without an interrupt due to any pending numeric exception. However, the brief assertion of FERR# is sufficient to latch the x87 FPU exception request into most hardware interface implementations (including Intel's recommended circuit).

All the x87 FPU instructions are implemented such that during their execution, there is a window in which the processor will sample and accept external interrupts. If there is a pending interrupt, the processor services the interrupt first before resuming the execution of the instruction. Consequently, it is possible that the no-wait floating-point instruction may accept the external interrupt caused by it's own assertion of the FERR# pin in the event of a pending unmasked numeric exception, which is not an explicitly documented behavior of a no-wait instruction. This process is illustrated in Figure D-3.

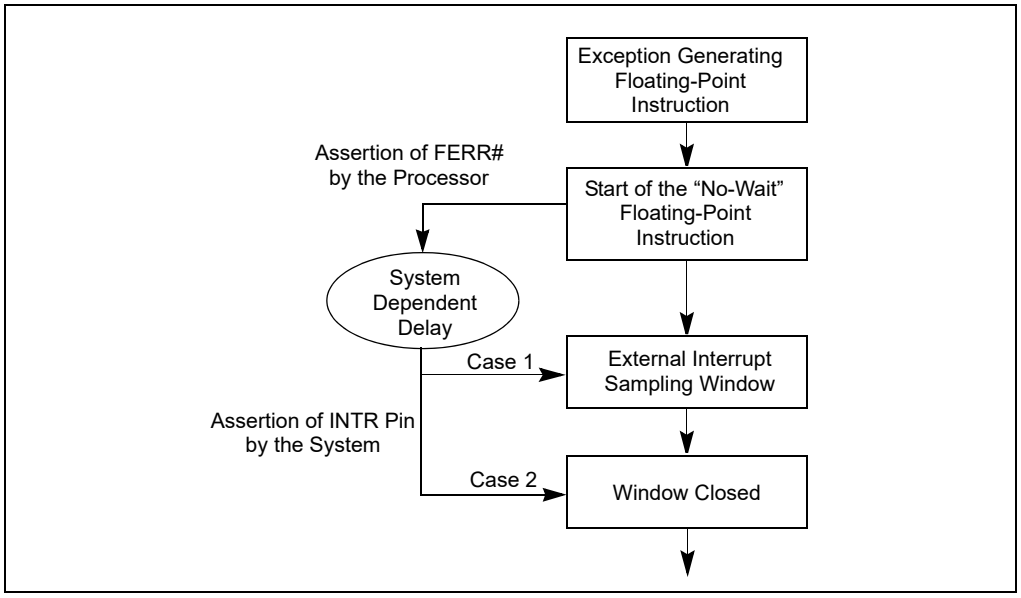


Figure D-3. Timing of Receipt of External Interrupt

Figure D-3 assumes that a floating-point instruction that generates a "deferred" error (as defined in the Section D.2.1.1, "Basic Rules: When FERR# Is Generated"), which asserts the FERR# pin only on encountering the next floating-point instruction, causes an unmasked numeric exception. Assume that the next floating-point instruction following this instruction is one of the no-wait floating-point instructions. The FERR# pin is asserted by the processor to indicate the pending exception on encountering the no-wait floating-point instruction. After the assertion of the FERR# pin the no-wait floating-point instruction opens a window where the pending external interrupts are sampled.

Then there are two cases possible depending on the timing of the receipt of the interrupt via the INTR pin (asserted by the system in response to the FERR# pin) by the processor.

Case 1 If the system responds to the assertion of FERR# pin by the no-wait floating-point instruction via the INTR pin during this window then the interrupt is serviced first, before resuming the execution of the no-wait floating-point instruction.

Case 2 If the system responds via the INTR pin after the window has closed then the interrupt is recognized only at the next instruction boundary.

There are two other ways, in addition to Case 1 above, in which a no-wait floating-point instruction can service a numeric exception inside its interrupt window. First, the first floating-point error condition could be of the "immediate" category (as defined in Section D.2.1.1, "Basic Rules: When FERR# Is Generated") that asserts FERR# immediately. If the system delay before asserting INTR is long enough, relative to the time elapsed before the no-wait floating-point instruction, INTR can be asserted inside the interrupt window for the latter. Second, consider two no-wait x87 FPU instructions in close sequence, and assume that a previous x87 FPU instruction has caused an unmasked numeric exception. Then if the INTR timing is too long for an FERR# signal triggered by the first no-wait instruction to hit the first instruction's interrupt window, it could catch the interrupt window of the second.

The possible malfunction of a no-wait x87 FPU instruction explained above cannot happen if the instruction is being used in the manner for which Intel originally designed it. The no-wait instructions were intended to be used inside the x87 FPU exception handler, to allow manipulation of the x87 FPU before the error condition is cleared, without hanging the processor because of the x87 FPU error condition, and without the need to assert IGNNE#. They will perform this function correctly, since before the error condition is cleared, the assertion of FERR# that caused the x87 FPU error handler to be invoked is still active. Thus the logic that would assert FERR# briefly at a no-wait instruction causes no change since FERR# is already asserted. The no-wait instructions may also be used without problem in the handler after the error condition is cleared, since now they will not cause FERR# to be asserted at all.

If a no-wait instruction is used outside of the x87 FPU exception handler, it may malfunction as explained above, depending on the details of the hardware interface implementation and which particular processor is involved. The actual interrupt inside the window in the no-wait instruction may be blocked by surrounding it with the instructions: PUSHFD, CLI, no-wait, then POPFD. (CLI blocks interrupts, and the push and pop of flags preserves and restores the original value of the interrupt flag.) However, if FERR# was triggered by the no-wait, its latched value and the PIC response will still be in effect. Further code can be used to check for and correct such a condition, if needed. Section D.3.6, "Considerations When x87 FPU Shared Between Tasks," discusses an important example of this type of problem and gives a solution.

D.2.2 MS-DOS* Compatibility Sub-mode in the P6 Family and Pentium® 4 Processors

When bit NE = 0 in CR0, the MS-DOS compatibility mode of the P6 family and Pentium 4 processors provides FERR# and IGNNE# functionality that is almost identical to the Intel486 and Pentium processors. The same external hardware described in Section D.2.1.2, "Recommended External Hardware to Support the MS-DOS* Compatibility Sub-mode," is recommended for the P6 family and Pentium 4 processors as well as the two previous generations. The only change to MS-DOS compatibility x87 FPU exception handling with the P6 family and Pentium 4 processors is that all exceptions for all x87 FPU instructions cause immediate error reporting. That is, FERR# is asserted as soon as the x87 FPU detects an unmasked exception; there are no cases in which error reporting is deferred to the next x87 FPU or WAIT instruction.

(As is discussed in Section D.2.1.1, "Basic Rules: When FERR# Is Generated," most exception cases in the Intel486 and Pentium processors are of the deferred type.)

Although FERR# is asserted immediately upon detection of an unmasked x87 FPU error, this certainly does not mean that the requested interrupt will always be serviced before the next instruction in the code sequence is executed. To begin with, the P6 family and Pentium 4 processors execute several instructions simultaneously. There also will be a delay, which depends on the external hardware implementation, between the FERR# assertion from the processor and the responding INTR assertion to the processor. Further, the interrupt request to the PICs (IRQ13) may be temporarily blocked by the operating system, or delayed by higher priority interrupts, and processor response to INTR itself is blocked if the operating system has cleared the IF bit in EFLAGS. Note that Streaming SIMD Extensions numeric exceptions will not cause assertion of FERR# (independent of the value of CR0.NE). In addition, they ignore the assertion/deassertion of IGNNE#).

However, just as with the Intel486 and Pentium processors, if the IGNNE# input is inactive, a floating-point exception which occurred in the previous x87 FPU instruction and is unmasked causes the processor to freeze immediately when encountering the next WAIT or x87 FPU instruction (except for no-wait instructions). This means that if the x87 FPU exception handler has not already been invoked due to the earlier exception (and therefore, the handler not has cleared that exception state from the x87 FPU), the processor is forced to wait for the handler to be invoked and handle the exception, before the processor can execute another WAIT or x87 FPU instruction.

As explained in Section D.2.1.3, "No-Wait x87 FPU Instructions Can Get x87 FPU Interrupt in Window," if a no-wait instruction is used outside of the x87 FPU exception handler, in the Intel486 and Pentium processors, it may accept an unmasked exception from a previous x87 FPU instruction which happens to fall within the external interrupt sampling window that is opened near the beginning of execution of all x87 FPU instructions. This will not happen in the P6 family and Pentium 4 processors, because this sampling window has been removed from the no-wait group of x87 FPU instructions.

D.3 RECOMMENDED PROTOCOL FOR MS-DOS* COMPATIBILITY HANDLERS

The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities such as deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply adds, subtracts, multiplies, and performs other operations on the numeric operands. The processor is designed to handle these two parts separately and efficiently. An x87 FPU exception handler, if a system chooses to implement one, is often one of the most complicated parts of the program control code.

D.3.1 Floating-Point Exceptions and Their Defaults

The x87 FPU can recognize six classes of floating-point exception conditions while executing floating-point instructions:

1. **#I** — Invalid operation
   #IS — Stack fault
   #IA — IEEE standard invalid operation
2. **#Z** — Divide-by-zero
3. **#D** — Denormalized operand
4. **#O** — Numeric overflow
5. **#U** — Numeric underflow
6. **#P** — Inexact result (precision)

For complete details on these exceptions and their defaults, see Section 8.4, "x87 FPU Floating-Point Exception Handling," and Section 8.5, "x87 FPU Floating-Point Exception Conditions."

D.3.2 Two Options for Handling Numeric Exceptions

Depending on options determined by the software system designer, the processor takes one of two possible courses of action when a numeric exception occurs:

1. The x87 FPU can handle selected exceptions itself, producing a default fix-up that is reasonable in most situations. This allows the numeric program execution to continue undisturbed. Programs can mask individual exception types to indicate that the x87 FPU should generate this safe, reasonable result whenever the exception occurs. The default exception fix-up activity is treated by the x87 FPU as part of the instruction causing the exception; no external indication of the exception is given (except that the instruction takes longer to execute when it handles a masked exception.) When masked exceptions are detected, a flag is set in the numeric status register, but no information is preserved regarding where or when it was set.
2. A software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the x87 FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. The exception handler can then implement any sort of recovery procedures desired for any numeric exception detectable by the x87 FPU.

D.3.2.1 Automatic Exception Handling: Using Masked Exceptions

Each of the six exception conditions described above has a corresponding flag bit in the x87 FPU status word and a mask bit in the x87 FPU control word. If an exception is masked (the corresponding mask bit in the control word = 1), the processor takes an appropriate default action and continues with the computation.

The processor has a default fix-up activity for every possible exception condition it may encounter. These masked-exception responses are designed to be safe and are generally acceptable for most numeric applications.

For example, if the Inexact result (Precision) exception is masked, the system can specify whether the x87 FPU should handle a result that cannot be represented exactly by one of four modes of rounding: rounding it normally, chopping it toward zero, always rounding it up, or always down. If the Underflow exception is masked, the x87 FPU will store a number that is too small to be represented in normalized form as a denormal (or zero if it's smaller than the smallest denormal). Note that when exceptions are masked, the x87 FPU may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the x87 FPU could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

As an example of how even severe exceptions can be handled safely and automatically using the default exception responses, consider a calculation of the parallel resistance of several values using only the standard formula (see Figure D-4). If R1 becomes zero, the circuit resistance becomes zero. With the divide-by-zero and precision exceptions masked, the processor will produce the correct result. FDIV of R1 into 1 gives infinity, and then FDIV of (infinity +R2 +R3) into 1 gives zero.

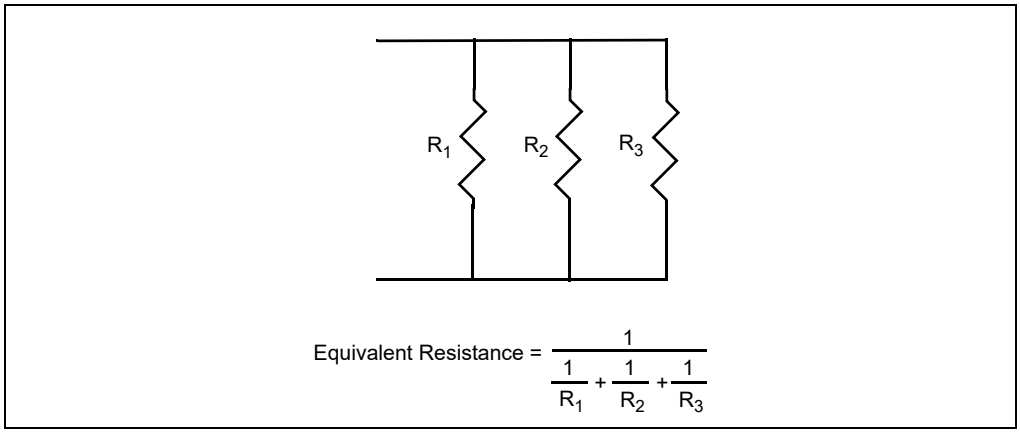


Figure D-4. Arithmetic Example Using Infinity

By masking or unmasking specific numeric exceptions in the x87 FPU control word, programmers can delegate responsibility for most exceptions to the processor, reserving the most severe exceptions for programmed exception handlers. Exception-handling software is often difficult to write, and the masked responses have been tailored to deliver the most reasonable result for each condition. For the majority of applications, masking all exceptions yields satisfactory results with the least programming effort. Certain exceptions can usefully be left unmasked during the debugging phase of software development, and then masked when the clean software is actually run. An invalid-operation exception for example, typically indicates a program error that must be corrected.

The exception flags in the x87 FPU status word provide a cumulative record of exceptions that have occurred since these flags were last cleared. Once set, these flags can be cleared only by executing the FCLEX/FNCLEX (clear exceptions) instruction, by reinitializing the x87 FPU with FINIT/FNINIT or FSAVE/FNSAVE, or by overwriting the flags with an FRSTOR or FLDENV instruction. This allows a programmer to mask all exceptions, run a calculation, and then inspect the status word to see if any exceptions were detected at any point in the calculation.

D.3.2.2 Software Exception Handling

If the x87 FPU in or with an IA-32 processor (Intel 286 and onwards) encounters an unmasked exception condition, with the system operated in the MS-DOS compatibility mode and with IGNNE# not asserted, a software exception handler is invoked through a PIC and the processor's INTR pin. The FERR# (or ERROR#) output from the x87 FPU that begins the process of invoking the exception handler may occur when the error condition is first detected, or when the processor encounters the next WAIT or x87 FPU instruction. Which of these two cases occurs depends on the processor generation and also on which exception and which x87 FPU instruction triggered it, as discussed earlier in Section D.1, "MS-DOS Compatibility Sub-mode for Handling x87 FPU Exceptions," and Section D.2, "Implementation of the MS-DOS* Compatibility Sub-mode in the Intel486™, Pentium®, and P6 Processor Family, and Pentium® 4 Processors." The elapsed time between the initial error signal and the invocation of the x87 FPU exception handler depends of course on the external hardware interface, and also on whether the external interrupt for x87 FPU errors is enabled. But the architecture ensures that the handler will be invoked before execution of the next WAIT or floating-point instruction since an unmasked floating-point exception causes the processor to freeze just before executing such an instruction (unless the IGNNE# input is active, or it is a no-wait x87 FPU instruction).

The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# (or ERROR#) output of the processor (or coprocessor), usually through IRQ13 on the "slave" PIC, and then through INTR. Then the external interrupt invokes the exception handling routine. Note that if the external interrupt for x87 FPU errors is disabled when the processor executes an x87 FPU instruction, the processor will freeze until some other (enabled) interrupt occurs if an unmasked x87 FPU exception condition is in effect. If NE = 0 but the IGNNE# input is active, the processor disregards the exception and continues. Error reporting via an external interrupt is supported for MS-DOS compatibility. Chapter 22, "IA-32 Architecture Compatibility," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* contains further discussion of compatibility issues.

The references above to the ERROR# output from the x87 FPU apply to the Intel 387 and Intel 287 math coprocessors (Numeric Processor Extension, or NPX, chips). If one of these coprocessors encounters an unmasked exception condition, it signals the exception to the Intel 286 or Intel386 processor using the ERROR# status line between the processor and the coprocessor. See Section D.1, "MS-DOS Compatibility Sub-mode for Handling x87 FPU Exceptions," in this appendix, and Chapter 22, "IA-32 Architecture Compatibility," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B,* for differences in x87 FPU exception handling.

The exception-handling routine is normally a part of the systems software. The routine must clear (or disable) the active exception flags in the x87 FPU status word before executing any floating-point instructions that cannot complete execution when there is a pending floating-point exception. Otherwise, the floating-point instruction will trigger the x87 FPU interrupt again, and the system will be caught in an endless loop of nested floating-point exceptions, and hang. In any event, the routine must clear (or disable) the active exception flags in the x87 FPU status word after handling them, and before IRET(D). Typical exception responses may include:

- Incrementing an exception counter for later display or printing.
- Printing or displaying diagnostic information (e.g., the x87 FPU environment and registers).
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it.

Applications programmers should consult their operating system's reference manuals for the appropriate system response to numerical exceptions. For systems programmers, some details on writing software exception handlers are provided in Chapter 6, "Interrupt and Exception Handling," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* as well as in Section D.3.4, "x87 FPU Exception Handling Examples," in this appendix.

As discussed in Section D.2.1.2, "Recommended External Hardware to Support the MS-DOS* Compatibility Sub-mode," some early FERR# to INTR hardware interface implementations are less robust than the recommended circuit. This is because they depended on the exception handler to clear the x87 FPU exception interrupt request to the PIC (by accessing port 0F0H) **before** the handler causes FERR# to be de-asserted by clearing the exception from the x87 FPU itself. To eliminate the chance of a problem with this early hardware, Intel recommends that x87 FPU exception handlers always access port 0F0H before clearing the error condition from the x87 FPU.

D.3.3 Synchronization Required for Use of x87 FPU Exception Handlers

Concurrency or synchronization management requires a check for exceptions before letting the processor change a value just used by the x87 FPU. It is important to remember that almost any numeric instruction can, under the wrong circumstances, produce a numeric exception.

D.3.3.1 Exception Synchronization: What, Why, and When

Exception synchronization means that the exception handler inspects and deals with the exception in the context in which it occurred. If concurrent execution is allowed, the state of the processor when it recognizes the exception is often **not** in the context in which it occurred. The processor may have changed many of its internal registers and be executing a totally different program by the time the exception occurs. If the exception handler cannot recapture the original context, it cannot reliably determine the cause of the exception or recover successfully from the exception. To handle this situation, the x87 FPU has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted.

This provides tools to help the exception handler recapture the original context, but the application code must also be written with synchronization in mind. Overall, exception synchronization must ensure that the x87 FPU and other relevant parts of the context are in a well defined state when the handler is invoked after an unmasked numeric exception occurs.

When the x87 FPU signals an unmasked exception condition, it is requesting help. The fact that the exception was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the x87 FPU will probably yield invalid results. Thus the exception must be handled, and with proper synchronization, or the program will not operate reliably.

For programmers using higher-level languages, all required synchronization is automatically provided by the appropriate compiler. However, for assembly language programmers exception synchronization remains the responsibility of the programmer. It is not uncommon for a programmer to expect that their numeric program will

not cause numeric exceptions after it has been tested and debugged, but in a different system or numeric environment, exceptions may occur regularly nonetheless. An obvious example would be use of the program with some numbers beyond the range for which it was designed and tested. Example D-1 and Example D-2 in Section D.3.3.2, "Exception Synchronization Examples," show a subtle way in which unexpected exceptions can occur.

As described in Section D.3.1, "Floating-Point Exceptions and Their Defaults," depending on options determined by the software system designer, the processor can perform one of two possible courses of action when a numeric exception occurs.

- The x87 FPU can provide a default fix-up for selected numeric exceptions. If the x87 FPU performs its default action for all exceptions, then the need for exception synchronization is not manifest. However, code is often ported to contexts and operating systems for which it was not originally designed. Example D-1 and Example D-2, below, illustrate that it is safest to always consider exception synchronization when designing code that uses the x87 FPU.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the x87 FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. When an x87 FPU exception handler will be invoked, synchronization must always be considered to assure reliable performance.

Example D-1 and Example D-2, below, illustrate the need to always consider exception synchronization when writing numeric code, even when the code is initially intended for execution with exceptions masked.

D.3.3.2 Exception Synchronization Examples

In the following examples, three instructions are shown to load an integer, calculate its square root, then increment the integer. The synchronous execution of the x87 FPU will allow both of these programs to execute correctly, with INC COUNT being executed in parallel in the processor, as long as no exceptions occur on the FILD instruction. However, if the code is later moved to an environment where exceptions are unmasked, the code in Example D-1 will not work correctly:

Example D-1. Incorrect Error Synchronization

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

Example D-2. Proper Error Synchronization

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

In some operating systems supporting the x87 FPU, the numeric register stack is extended to memory. To extend the x87 FPU stack to memory, the invalid exception is unmasked. A push to a full register or pop from an empty register sets SF (Stack Fault flag) and causes an invalid operation exception. The recovery routine for the exception must recognize this situation, fix up the stack, then perform the original operation. The recovery routine will not work correctly in Example D-1. The problem is that the value of COUNT increments before the exception handler is invoked, so that the recovery routine will load an incorrect value of COUNT, causing the program to fail or behave unreliably.

D.3.3.3 Proper Exception Synchronization

As explained in Section D.2.1.2, "Recommended External Hardware to Support the MS-DOS* Compatibility Submode," if the x87 FPU encounters an unmasked exception condition a software exception handler is invoked **before** execution of the **next** WAIT or floating-point instruction. This is because an unmasked floating-point exception causes the processor to freeze immediately before executing such an instruction (unless the IGNNE# input is

active, or it is a no-wait x87 FPU instruction). Exactly when the exception handler will be invoked (in the interval between when the exception is detected and the next WAIT or x87 FPU instruction) is dependent on the processor generation, the system, and which x87 FPU instruction and exception is involved.

To be safe in exception synchronization, one should assume the handler will be invoked at the end of the interval. Thus the program should not change any value that might be needed by the handler (such as COUNT in Example D-1 and Example D-2) until **after** the **next** x87 FPU instruction following an x87 FPU instruction that could cause an error. If the program needs to modify such a value before the next x87 FPU instruction (or if the next x87 FPU instruction could also cause an error), then a WAIT instruction should be inserted before the value is modified. This will force the handling of any exception before the value is modified. A WAIT instruction should also be placed after the last floating-point instruction in an application so that any unmasked exceptions will be serviced before the task completes.

D.3.4 x87 FPU Exception Handling Examples

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler procedure as consisting of "prologue," "body," and "epilogue" sections of code.

In the transfer of control to the exception handler due to an INTR, NMI, or SMI, external interrupts have been disabled by hardware. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically, this involves saving registers and transferring diagnostic information from the x87 FPU to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler. The standard "prologue" not only saves the registers and transfers diagnostic information from the x87 FPU to memory but also clears the floating-point exception flags in the status word. Alternatively, when it is not necessary for the handler to be re-entrant, another technique may also be used. In this technique, the exception flags are not cleared in the "prologue" and the body of the handler must not contain any floating-point instructions that cannot complete execution when there is a pending floating-point exception. (The no-wait instructions are discussed in Section 8.3.12, "Waiting vs. Non-waiting Instructions.") Note that the handler must still clear the exception flag(s) before executing the IRET. If the exception handler uses neither of these techniques, the system will be caught in an endless loop of nested floating-point exceptions, and hang.

The body of the exception handler examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution. The epilogue essentially reverses the actions of the prologue, restoring the processor so that normal execution can be resumed. The epilogue must not load an unmasked exception flag into the x87 FPU or another exception will be requested immediately.

The following code examples show the ASM386/486 coding of three skeleton exception handlers, with the save spaces given as correct for 32-bit protected mode. They show how prologues and epilogues can be written for various situations, but the application-dependent exception handling body is just indicated by comments showing where it should be placed.

The first two are very similar; their only substantial difference is their choice of instructions to save and restore the x87 FPU. The trade-off here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. (Also, after saving the original contents, FNSAVE re-initializes the x87 FPU, while FNSTENV only masks all x87 FPU exceptions.) For applications that are sensitive to interrupt latency or that do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the processor does not recognize another interrupt request. (See the Section 8.1.10, "Saving the x87 FPU's State with FSTENV/FNSTENV and FSAVE/FNSAVE," for a complete description of the x87 FPU save image.) If the processor supports Streaming SIMD Extensions and the operating system supports it, the FXSAVE instruction should be used instead of FNSAVE. If the FXSAVE instruction is used, the save area should be increased to 512 bytes and aligned to 16 bytes to save the entire state. These steps will ensure that the complete context is saved.

After the exception handler body, the epilogues prepare the processor to resume execution from the point of interruption (for example, the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the x87 FPU are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared).

Example D-3 and Example D-4 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in Example D-5 can be employed. The basic technique is to

save the full x87 FPU state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

Example D-3. Full-State Exception Handler

```
SAVE_ALL    PROC
;
;SAVE REGISTERS, ALLOCATE STACK SPACE FOR x87 FPU STATE IMAGE
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE FULL x87 FPU STATE, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSAVE  [EBP-108]
    PUSH    [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
    POPFD   ;RESTORE IF TO VALUE BEFORE x87 FPU EXCEPTION
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
;RESTORE MODIFIED STATE IMAGE
    MOV     BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
;DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV     ESP, EBP
    .
    .
    POP     EBP
;
;RETURN TO INTERRUPTED CALCULATION
    IRETD
    SAVE_ALL    ENDP
```

Example D-4. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENTPROC
;
;SAVE REGISTERS, ALLOCATE STACK SPACE FOR x87 FPU ENVIRONMENT
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 28   ;ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSTENV     [EBP - 28]
    PUSH        [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
    POPFD   ;RESTORE IF TO VALUE BEFORE x87 FPU EXCEPTION
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
;RESTORE MODIFIED ENVIRONMENT IMAGE
    MOV     BYTE PTR [EBP-24], 0H
```

```
    FLDENV  [EBP-28]
;DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV     ESP, EBP
    .
    .
    POP     EBP
;
;RETURN TO INTERRUPTED CALCULATION
    IRETD
    SAVE_ENVIRONMENT ENDP
```

Example D-5. Reentrant Exception Handler

```
    .
    .
LOCAL_CONTROL DW ?; ASSUME INITIALIZED
    .
    .
REENTRANTPROC
;
;SAVE REGISTERS, ALLOCATE STACK SPACE FOR x87 FPU STATE IMAGE
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 108  ;ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)

;SAVE STATE, LOAD NEW CONTROL WORD, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSAVE  [EBP-108]
    FLDCW   LOCAL_CONTROL
    PUSH    [EBP + OFFSET_TO_EFLAGS]  ;COPY OLD EFLAGS TO STACK TOP
    POPFD   ;RESTORE IF TO VALUE BEFORE x87 FPU EXCEPTION

    .
    .
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE
;GOES HERE - AN UNMASKED EXCEPTION
;GENERATED HERE WILL CAUSE THE EXCEPTION HANDLER TO BE REENTERED
;IF LOCAL STORAGE IS NEEDED, IT MUST BE ALLOCATED ON THE STACK
    .
;CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
;RESTORE MODIFIED STATE IMAGE
    MOV     BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
;DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV     ESP, EBP
    .
    .
    POP     EBP
;
;RETURN TO POINT OF INTERRUPTION
    IRETD
    REENTRANT ENDP
```

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

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

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

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

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

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

To avoid this problem, Intel recommends two measures:

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

D.3.6 Considerations When x87 FPU Shared Between Tasks

The IA-32 architecture allows speculative deferral of floating-point state swaps on task switches. This feature allows postponing an x87 FPU state swap until an x87 FPU instruction is actually encountered in another task. Since kernel tasks rarely use floating-point, and some applications do not use floating-point or use it infrequently, the amount of time saved by avoiding unnecessary stores of the floating-point state is significant. Speculative deferral of x87 FPU saves does, however, place an extra burden on the kernel in three key ways:

1. The kernel must keep track of which thread owns the x87 FPU, which may be different from the currently executing thread.
2. The kernel must associate any floating-point exceptions with the generating task. This requires special handling since floating-point exceptions are delivered asynchronous with other system activity.

3. There are conditions under which spurious floating-point exception interrupts are generated, which the kernel must recognize and discard.

D.3.6.1 Speculatively Deferring x87 FPU Saves, General Overview

In order to support multitasking, each thread in the system needs a save area for the general-purpose registers, and each task that is allowed to use floating-point needs an x87 FPU save area large enough to hold the entire x87 FPU stack and associated x87 FPU state such as the control word and status word. (See Section 8.1.10, "Saving the x87 FPU's State with FSTENV/FNSTENV and FSAVE/FNSAVE," for a complete description of the x87 FPU save image.) If the processor and the operating system support Streaming SIMD Extensions, the save area should be large enough and aligned correctly to hold x87 FPU and Streaming SIMD Extensions state.

On a task switch, the general-purpose registers are swapped out to their save area for the suspending thread, and the registers of the resuming thread are loaded. The x87 FPU state does not need to be saved at this point. If the resuming thread does not use the x87 FPU before it is itself suspended, then both a save and a load of the x87 FPU state has been avoided. It is often the case that several threads may be executed without any usage of the x87 FPU.

The processor supports speculative deferral of x87 FPU saves via interrupt 7 "Device Not Available" (DNA), used in conjunction with CR0 bit 3, the "Task Switched" bit (TS). (See "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.*) Every task switch via the hardware supported task switching mechanism (see "Task Switching" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A)* sets TS. Multi-threaded kernels that use software task switching[1] can set the TS bit by reading CR0, ORing a "1" into[2] bit 3, and writing back CR0. Any subsequent floating-point instructions (now being executed in a new thread context) will fault via interrupt 7 before execution.

This allows a DNA handler to save the old floating-point context and reload the x87 FPU state for the current thread. The handler should clear the TS bit before exit using the CLTS instruction. On return from the handler the faulting thread will proceed with its floating-point computation.

Some operating systems save the x87 FPU context on every task switch, typically because they also change the linear address space between tasks. The problem and solution discussed in the following sections apply to these operating systems also.

D.3.6.2 Tracking x87 FPU Ownership

Since the contents of the x87 FPU may not belong to the currently executing thread, the thread identifier for the last x87 FPU user needs to be tracked separately. This is not complicated; the kernel should simply provide a variable to store the thread identifier of the x87 FPU owner, separate from the variable that stores the identifier for the currently executing thread. This variable is updated in the DNA exception handler, and is used by the DNA exception handler to find the x87 FPU save areas of the old and new threads. A simplified flow for a DNA exception handler is then:

1. Use the "x87 FPU Owner" variable to find the x87 FPU save area of the last thread to use the x87 FPU.
2. Save the x87 FPU contents to the old thread's save area, typically using an FNSAVE or FXSAVE instruction.
3. Set the x87 FPU Owner variable to the identify the currently executing thread.
4. Reload the x87 FPU contents from the new thread's save area, typically using an FRSTOR or FXSTOR instruction.
5. Clear TS using the CLTS instruction and exit the DNA exception handler.

While this flow covers the basic requirements for speculatively deferred x87 FPU state swaps, there are some additional subtleties that need to be handled in a robust implementation.

---

1 In a software task switch, the operating system uses a sequence of instructions to save the suspending thread's state and restore the resuming thread's state, instead of the single long non-interruptible task switch operation provided by the IA-32 architecture.

2 Although CR0, bit 2, the emulation flag (EM), also causes a DNA exception, **do not** use the EM bit as a surrogate for TS. EM means that no x87 FPU is available and that floating-point instructions must be emulated. Using EM to trap on task switches is not compatible with the MMX technology. If the EM flag is set, MMX instructions raise the invalid opcode exception.

D.3.6.3 Interaction of x87 FPU State Saves and Floating-Point Exception Association

Recall these key points from earlier in this document: When considering floating-point exceptions across all implementations of the IA-32 architecture, and across all floating-point instructions, a floating-point exception can be initiated from any time during the excepting floating-point instruction, up to just before the next floating-point instruction. The "next" floating-point instruction may be the FNSAVE used to save the x87 FPU state for a task switch. In the case of "no-wait:" instructions such as FNSAVE, the interrupt from a previously excepting instruction (NE = 0 case) may arrive just before the no-wait instruction, during, or shortly thereafter with a system dependent delay.

Note that this implies that an floating-point exception might be registered during the state swap process itself, and the kernel and floating-point exception interrupt handler must be prepared for this case.

A simple way to handle the case of exceptions arriving during x87 FPU state swaps is to allow the kernel to be one of the x87 FPU owning threads. A reserved thread identifier is used to indicate kernel ownership of the x87 FPU. During an floating-point state swap, the "x87 FPU owner" variable should be set to indicate the kernel as the current owner. At the completion of the state swap, the variable should be set to indicate the new owning thread. The numeric exception handler needs to check the x87 FPU owner and **discard** any numeric exceptions that occur while the kernel is the x87 FPU owner. A more general flow for a DNA exception handler that handles this case is shown in Figure D-5.

Numeric exceptions received while the kernel owns the x87 FPU for a state swap must be discarded in the kernel without being dispatched to a handler. A flow for a numeric exception dispatch routine is shown in Figure D-6.

It may at first glance seem that there is a possibility of floating-point exceptions being lost because of exceptions that are discarded during state swaps. This is not the case, as the exception will be re-issued when the floating-point state is reloaded. Walking through state swaps both with and without pending numeric exceptions will clarify the operation of these two handlers.

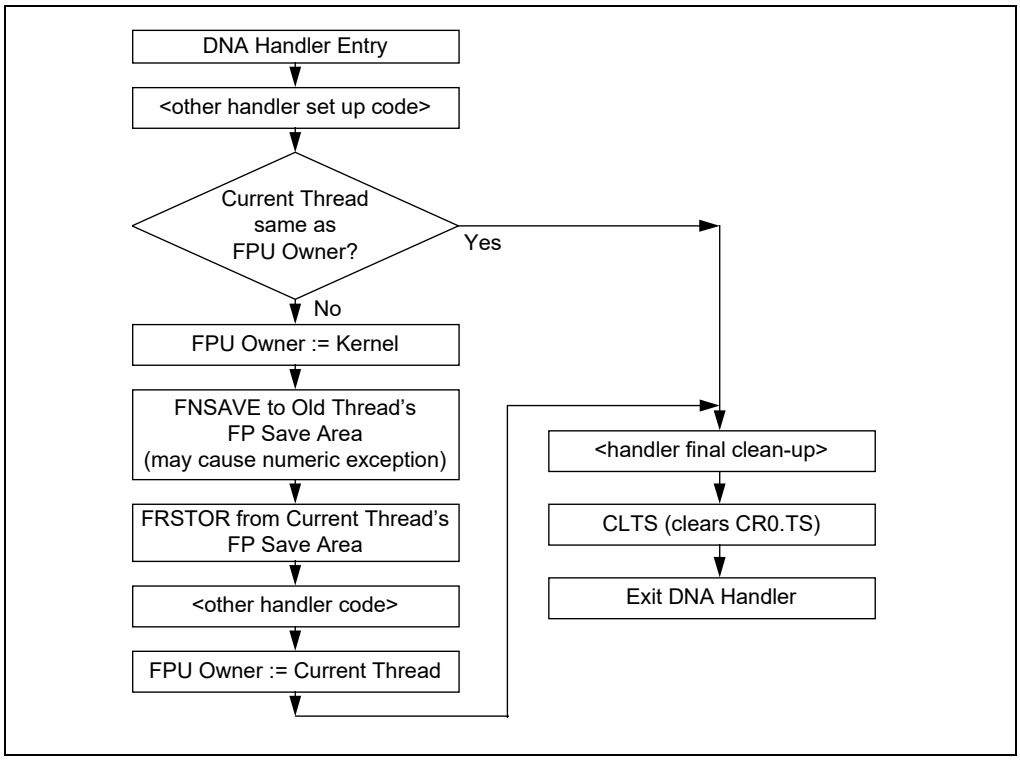


Figure D-5. General Program Flow for DNA Exception Handler

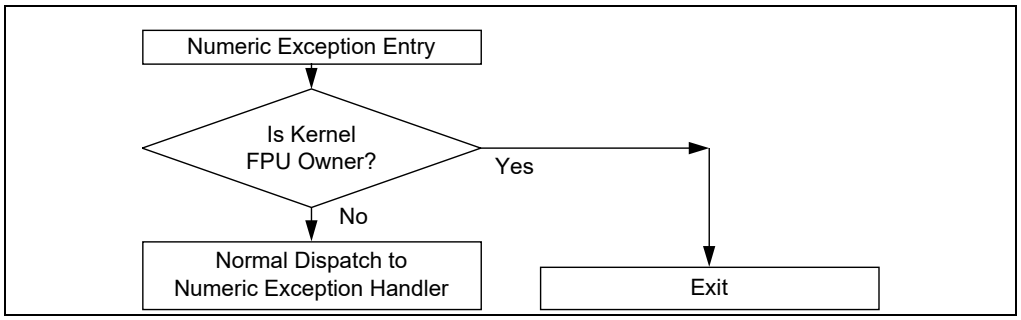


Figure D-6. Program Flow for a Numeric Exception Dispatch Routine

Case #1: x87 FPU State Swap Without Numeric Exception

Assume two threads A and B, both using the floating-point unit. Let A be the thread to have most recently executed a floating-point instruction, with no pending numeric exceptions. Let B be the currently executing thread. CR0.TS was set when thread A was suspended.

When B starts to execute a floating-point instruction the instruction will fault with the DNA exception because TS is set.

At this point the handler is entered, and eventually it finds that the current x87 FPU Owner is not the currently executing thread. To guard the x87 FPU state swap from extraneous numeric exceptions, the x87 FPU Owner is set to be the kernel. The old owner's x87 FPU state is saved with FNSAVE, and the current thread's x87 FPU state is restored with FRSTOR. Before exiting, the x87 FPU owner is set to thread B, and the TS bit is cleared.

On exit, thread B resumes execution of the faulting floating-point instruction and continues.

Case #2: x87 FPU State Swap with Discarded Numeric Exception

Again, assume two threads A and B, both using the floating-point unit. Let A be the thread to have most recently executed a floating-point instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a floating-point instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the x87 FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old floating-point context of thread A and the FRSTOR of the floating-point context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an floating-point instruction, once again the DNA exception handler is entered. B's x87 FPU state is saved with FNSAVE, and A's x87 FPU state is restored with FRSTOR. Note that in restoring the x87 FPU state from A's save area, the pending numeric exception flags are reloaded into the floating-point status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting floating-point instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting x87 FPU state swap via the DNA exception handler causes an extra numeric exception which can be safely discarded.

D.3.6.4 Interrupt Routing From the Kernel

In MS-DOS, an application that wishes to handle numeric exceptions hooks interrupt 16 by placing its handler address in the interrupt vector table, and exiting via a jump to the previous interrupt 16 handler. Protected mode systems that run MS-DOS programs under a subsystem can emulate this exception delivery mechanism. For example, assume a protected mode OS. that runs with CR0.NE[bit 5] = 1, and that runs MS-DOS programs in a

virtual machine subsystem. The MS-DOS program is set up in a virtual machine that provides a virtualized interrupt table. The MS-DOS application hooks interrupt 16 in the virtual machine in the normal way. A numeric exception will trap to the kernel via the real INT 16 residing in the kernel at ring 0.

The INT 16 handler in the kernel then locates the correct MS-DOS virtual machine, and reflects the interrupt to the virtual machine monitor. The virtual machine monitor then emulates an interrupt by jumping through the address in the virtualized interrupt table, eventually reaching the application's numeric exception handler.

D.3.6.5 Special Considerations for Operating Systems that Support Streaming SIMD Extensions

Operating systems that support Streaming SIMD Extensions instructions introduced with the Pentium III processor should use the FXSAVE and FXRSTOR instructions to save and restore the new SIMD floating-point instruction register state as well as the floating-point state. Such operating systems must consider the following issues:

1. **Enlarged state save area** — FNSAVE/FRSTOR instructions operate on a 94-byte or 108-byte memory region, depending on whether they are executed in 16-bit or 32-bit mode. The FXSAVE/FXRSTOR instructions operate on a 512-byte memory region.
2. **Alignment requirements** — FXSAVE/FXRSTOR instructions require the memory region on which they operate to be 16-byte aligned (refer to the individual instruction instructions descriptions in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for information about exceptions generated if the memory region is not aligned).
3. **Maintaining compatibility with legacy applications/libraries** — The operating system changes to support Streaming SIMD Extensions must be invisible to legacy applications or libraries that deal only with floating-point instructions. The layout of the memory region operated on by the FXSAVE/FXRSTOR instructions is different from the layout for the FNSAVE/FRSTOR instructions. Specifically, the format of the x87 FPU tag word and the length of the various fields in the memory region is different. Care must be taken to return the x87 FPU state to a legacy application (e.g., when reporting FP exceptions) in the format it expects.
4. **Instruction semantic differences** — There are some semantic differences between the way the FXSAVE and FSAVE/FNSAVE instructions operate. The FSAVE/FNSAVE instructions clear the x87 FPU after they save the state while the FXSAVE instruction saves the x87 FPU/Streaming SIMD Extensions state but does not clear it. Operating systems that use FXSAVE to save the x87 FPU state before making it available for another thread (e.g., during thread switch time) should take precautions not to pass a "dirty" x87 FPU to another application.

D.4 DIFFERENCES FOR HANDLERS USING NATIVE MODE

The 8087 has an INT pin which it asserts when an unmasked exception occurs. But there is no interrupt input pin in the 8086 or 8088 dedicated to its attachment, nor an interrupt vector in the 8086 or 8088 specific for an x87 FPU error assertion. Beginning with the Intel 286 and Intel 287 hardware, a connection was dedicated to support the x87 FPU exception and interrupt vector 16 was assigned to it.

D.4.1 Origin with the Intel 286 and Intel 287, and Intel386 and Intel 387 Processors

The Intel 286 and Intel 287, and Intel386 and Intel 387 processor/coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and x87 FPU. If this is done, when an unmasked x87 FPU exception occurs, the x87 FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line immediately before execution of the next WAIT or x87 FPU instruction (except for the no-wait type) in its instruction stream, and branches to the handler of interrupt 16. Thus an x87 FPU exception will be handled before any other x87 FPU instruction (after the one causing the error) is executed (except for no-wait instructions, which will be executed without triggering the x87 FPU exception interrupt, but it will remain pending).

Using the dedicated INT 16 for x87 FPU exception handling is referred to as the native mode. It is the simplest approach, and the one recommended most highly by Intel.

D.4.2 Changes with Intel486, Pentium and Pentium Pro Processors with CR0.NE[bit 5] = 1

With these three generations of the IA-32 architecture, more enhancements and speedup features have been added to the corresponding x87 FPUs. Also, the x87 FPU is now built into the same chip as the processor, which allows further increases in the speed at which the x87 FPU can operate as part of the integrated system. This also means that the native mode of x87 FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an x87 FPU instruction, the x87 FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or x87 FPU instruction (except for no-wait instructions, which will be executed as described in Section D.4.1, "Origin with the Intel 286 and Intel 287, and Intel386 and Intel 387 Processors").

An unmasked numerical exception causes the FERR# output to be activated even with NE = 1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an operating system using the native mode is actually running the system, it is the operating system's responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in Section D.2.1.3, "No-Wait x87 FPU Instructions Can Get x87 FPU Interrupt in Window," where for Intel486 and Pentium processors a no-wait x87 FPU instruction can get an x87 FPU exception.

D.4.3 Considerations When x87 FPU Shared Between Tasks Using Native Mode

The protocols recommended in Section D.3.6, "Considerations When x87 FPU Shared Between Tasks," for MS-DOS compatibility x87 FPU exception handlers that are shared between tasks may be used without change with the native mode. However, the protocols for a handler written specifically for native mode can be simplified, because the problem of a spurious floating-point exception interrupt occurring while the kernel is executing cannot happen in native mode.

The problem as actually found in practical code in a MS-DOS compatibility system happens when the DNA handler uses FNSAVE to switch x87 FPU contexts. If an x87 FPU exception is active, then FNSAVE triggers FERR# briefly, which usually will cause the x87 FPU exception handler to be invoked inside the DNA handler. In native mode, neither FNSAVE nor any other no-wait instructions can trigger interrupt 16. (As discussed above, FERR# gets asserted independent of the value of the NE bit, but when NE = 1, the operating system should not enable its path through the PIC.) Another possible (very rare) way a floating-point exception interrupt could occur while the kernel is executing is by an x87 FPU immediate exception case having its interrupt delayed by the external hardware until execution has switched to the kernel. This also cannot happen in native mode because there is no delay through external hardware.

Thus the native mode x87 FPU exception handler can omit the test to see if the kernel is the x87 FPU owner, and the DNA handler for a native mode system can omit the step of setting the kernel as the x87 FPU owner at the handler's beginning. Since however these simplifications are minor and save little code, it would be a reasonable and conservative habit (as long as the MS-DOS compatibility mode is widely used) to include these steps in all systems.

Note that the special DP (Dual Processing) mode for Pentium processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium, P6 family, or Pentium 4 processors, support x87 FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility mode for systems using more than one processor.

8. Updates to Appendix E, Volume 1

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

Change to this chapter: Correction to CVTPS2PD/CVTSS2SD and CVTPD2PS/CVTSD2SS rows in Table E-13 "#I - Invalid Operations".

APPENDIX E
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

See Section 11.5, "SSE, SSE2, and SSE3 Exceptions," for a detailed discussion of SIMD floating-point exceptions.

This appendix considers only SSE/SSE2/SSE3 instructions that can generate numeric (SIMD floating-point) exceptions, and gives an overview of the necessary support for handling such exceptions. This appendix does not address instructions that do not generate floating-point exceptions (such as RSQRTSS, RSQRTPS, RCPSS, or RCPPS), any x87 instructions, or any unlisted instruction.

For detailed information on which instructions generate numeric exceptions, and a listing of those exceptions, refer to Appendix C, "Floating-Point Exceptions Summary." Non-numeric exceptions are handled in a way similar to that for the standard IA-32 instructions.

E.1 TWO OPTIONS FOR HANDLING FLOATING-POINT EXCEPTIONS

Just as for x87 FPU floating-point exceptions, the processor takes one of two possible courses of action when an SSE/SSE2/SSE3 instruction raises a floating-point exception:

- If the exception being raised is masked (by setting the corresponding mask bit in the MXCSR to 1), then a default result is produced which is acceptable in most situations. No external indication of the exception is given, but the corresponding exception flags in the MXCSR are set and may be examined later. Note though that for packed operations, an exception flag that is set in the MXCSR will not tell which of the sub-operands caused the event to occur.
- If the exception being raised is not masked (by setting the corresponding mask bit in the MXCSR to 0), a software exception handler previously registered by the user with operating system support will be invoked through the SIMD floating-point exception (#XM, exception 19). This case is discussed below in Section E.2, "Software Exception Handling."

E.2 SOFTWARE EXCEPTION HANDLING

The #XM handler is usually part of the system software (the operating system kernel). Note that an interrupt descriptor table (IDT) entry must have been previously set up for exception 19 (refer to Chapter 6, "Interrupt and Exception Handling," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A)*. Some compilers use specific run-time libraries to assist in floating-point exception handling. If any x87 FPU floating-point operations are going to be performed that might raise floating-point exceptions, then the exception handling routine must either disable all floating-point exceptions (for example, loading a local control word with FLDCW), or it must be implemented as re-entrant (for the case of x87 FPU exceptions, refer to Example D-1 in Appendix D, "Guidelines for Writing x87 FPU Exception Handlers"). If this is not the case, the routine has to clear the status flags for x87 FPU exceptions or to mask all x87 FPU floating-point exceptions. For SIMD floating-point exceptions though, the exception flags in MXCSR do not have to be cleared, even if they remain unmasked (but they may still be cleared). Exceptions are in this case precise and occur immediately, and a SIMD floating-point exception status flag that is set when the corresponding exception is unmasked will not generate an exception.

Typical actions performed by this low-level exception handling routine are:

- Incrementing an exception counter for later display or printing
- Printing or displaying diagnostic information (e.g. the MXCSR and XMM registers)
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it
- Storing information about the exception in a data structure that will be passed to a higher level user exception handler

In most cases (and this applies also to SSE/SSE2/SSE3 instructions), there will be three main components of a low-level floating-point exception handler: a prologue, a body, and an epilogue.

The prologue performs functions that must be protected from possible interruption by higher-priority sources - typically saving registers and transferring diagnostic information from the processor to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler (assuming that the interrupt handler was called through an interrupt gate, meaning that the processor cleared the interrupt enable (IF) flag in the EFLAGS register - refer to Section 6.4.1, "Call and Return Operation for Interrupt or Exception Handling Procedures").

The body of the exception handler examines the diagnostic information and makes a response that is application-dependent. It may range from halting execution, to displaying a message, to attempting to fix the problem and then proceeding with normal execution, to setting up a data structure, calling a higher-level user exception handler and continuing execution upon return from it. This latter case will be assumed in Section E.4, "SIMD Floating-Point Exceptions and the IEEE Standard 754" below.

Finally, the epilogue essentially reverses the actions of the prologue, restoring the processor state so that normal execution can be resumed.

The following example represents a typical exception handler. To link it with Example E-2 that will follow in Section E.4.3, "Example SIMD Floating-Point Emulation Implementation," assume that the body of the handler (not shown here in detail) passes the saved state to a routine that will examine in turn all the sub-operands of the excepting instruction, invoking a user floating-point exception handler if a particular set of sub-operands raises an unmasked (enabled) exception, or emulating the instruction otherwise.

Example E-1. SIMD Floating-Point Exception Handler

```
SIMD_FP_EXC_HANDLER PROC

;PROLOGUE
;SAVE REGISTERS THAT MIGHT BE USED BY THE EXCEPTION HANDLER
   PUSH EBP                          ;SAVE EBP
   PUSH EAX                          ;SAVE EAX
   ...
   MOV EBP, ESP                      ;SAVE ESP in EBP
   SUB ESP, 512                      ;ALLOCATE 512 BYTES
   AND ESP, 0fffffff0h               ;MAKE THE ADDRESS 16-BYTE ALIGNED
   FXSAVE [ESP]                      ;SAVE FP, MMX, AND SIMD FP STATE
   PUSH [EBP+EFLAGS_OFFSET]          ;COPY OLD EFLAGS TO STACK TOP
   POPFD                             ;RESTORE THE INTERRUPT ENABLE FLAG IF
                                     ;TO VALUE BEFORE SIMD FP EXCEPTION

;BODY
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
   LDMXCSR LOCAL_MXCSR               ;LOAD LOCAL MXCSR VALUE IF NEEDED
   ...
   ...
;EPILOGUE
   FXRSTOR [ESP]                     ;RESTORE MODIFIED STATE IMAGE
   MOV ESP, EBP                      ;DE-ALLOCATE STACK SPACE
   ...
   POP EAX                           ;RESTORE EAX
   POP EBP                           ;RESTORE EBP
   IRET                              ;RETURN TO INTERRUPTED CALCULATION
SIMD_FP_EXC_HANDLER ENDP
```

E.3 EXCEPTION SYNCHRONIZATION

An SSE/SSE2/SSE3 instruction can execute in parallel with other similar instructions, with integer instructions, and with floating-point or MMX instructions. Unlike for x87 instructions, special precaution for exception synchronization is not necessary in this case. This is because floating-point exceptions for SSE/SSE2/SSE3 instructions occur immediately and are not delayed until a subsequent floating-point instruction is executed. However, floating-point emulation may be necessary when unmasked floating-point exceptions are generated.

E.4 SIMD FLOATING-POINT EXCEPTIONS AND THE IEEE STANDARD 754

SSE/SSE2/SSE3 extensions are 100% compatible with the IEEE Standard 754 for Binary Floating-Point Arithmetic, satisfying all of its mandatory requirements (when the flush-to-zero or denormals-are-zeros modes are not enabled). But a programming environment that includes SSE/SSE2/SSE3 instructions will comply with both the obligatory and the strongly recommended requirements of the IEEE Standard 754 regarding floating-point exception handling, only as a combination of hardware and software (which is acceptable). The standard states that a user should be able to request a trap on any of the five floating-point exceptions (note that the denormal exception is an IA-32 addition), and it also specifies the values (operands or result) to be delivered to the exception handler.

The main issue is that for SSE/SSE2/SSE3 instructions that raise post-computation exceptions (traps: overflow, underflow, or inexact), unlike for x87 FPU instructions, the processor does not provide the result recommended by IEEE Standard 754 to the user handler. If a user program needs the result of an instruction that generated a post-computation exception, it is the responsibility of the software to produce this result by emulating the faulting SSE/SSE2/SSE3 instruction. Another issue is that the standard does not specify explicitly how to handle multiple floating-point exceptions that occur simultaneously. For packed operations, a logical OR of the flags that would be set by each sub-operation is used to set the exception flags in the MXCSR. The following subsections present one possible way to solve these problems.

E.4.1 Floating-Point Emulation

Every operating system must provide a kernel level floating-point exception handler (a template was presented in Section E.2, “Software Exception Handling” above). In the following discussion, assume that a user mode floating-point exception filter is supplied for SIMD floating-point exceptions (for example as part of a library of C functions), that a user program can invoke in order to handle unmasked exceptions. The user mode floating-point exception filter (not shown here) has to be able to emulate the subset of SSE/SSE2/SSE3 instructions that can generate numeric exceptions, and has to be able to invoke a user provided floating-point exception handler for floating-point exceptions. When a floating-point exception that is not masked is raised by an SSE/SSE2/SSE3 instruction, the low-level floating-point exception handler will be called. This low-level handler may in turn call the user mode floating-point exception filter. The filter function receives the original operands of the excepting instruction as no results are provided by the hardware, whether a pre-computation or a post-computation exception has occurred. The filter will unpack the operands into up to four sets of sub-operands, and will submit them one set at a time to an emulation function (See Example E-2 in Section E.4.3, “Example SIMD Floating-Point Emulation Implementation”). The emulation function will examine the sub-operands, and will possibly redo the necessary calculation.

Two cases are possible:

- If an unmasked (enabled) exception would occur in this process, the emulation function will return to its caller (the filter function) with the appropriate information. The filter will invoke a (previously registered) user floating-point exception handler for this set of sub-operands, and will record the result upon return from the user handler (provided the user handler allows continuation of the execution).
- If no unmasked (enabled) exception would occur, the emulation function will determine and will return to its caller the result of the operation for the current set of sub-operands (it has to be IEEE Standard 754 compliant). The filter function will record the result (plus any new flag settings).

The user level filter function will then call the emulation function for the next set of sub-operands (if any). When done with all the operand sets, the partial results will be packed (if the excepting instruction has a packed floating-point result, which is true for most SSE/SSE2/SSE3 numeric instructions) and the filter will return to the low-level exception handler, which in turn will return from the interruption, allowing execution to continue. Note that the

instruction pointer (EIP) has to be altered to point to the instruction following the excepting instruction, in order to continue execution correctly.

If a user mode floating-point exception filter is not provided, then all the work for decoding the excepting instruction, reading its operands, emulating the instruction for the components of the result that do not correspond to unmasked floating-point exceptions, and providing the compounded result will have to be performed by the user-provided floating-point exception handler.

Actual emulation might have to take place for one operand or pair of operands for scalar operations, and for all sub-operands or pairs of sub-operands for packed operations. The steps to perform are the following:

- The excepting instruction has to be decoded and the operands have to be read from the saved context.
- The instruction has to be emulated for each (pair of) sub-operand(s); if no floating-point exception occurs, the partial result has to be saved; if a masked floating-point exception occurs, the masked result has to be produced through emulation and saved, and the appropriate status flags have to be set; if an unmasked floating-point exception occurs, the result has to be generated by the user provided floating-point exception handler, and the appropriate status flags have to be set.
- The partial results have to be combined and written to the context that will be restored upon application program resumption.

A diagram of the control flow in handling an unmasked floating-point exception is presented below.

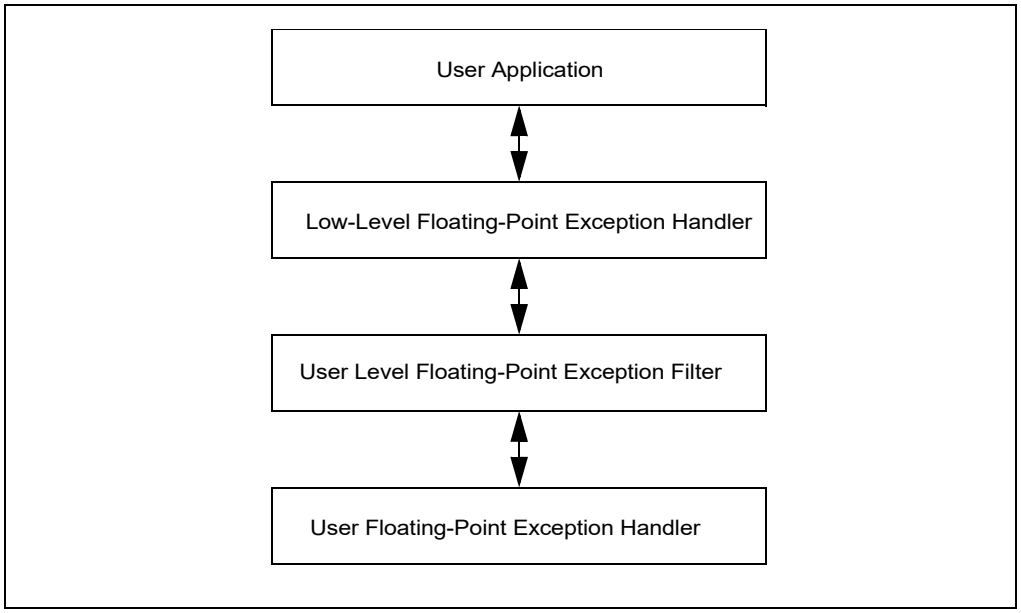


Figure E-1. Control Flow for Handling Unmasked Floating-Point Exceptions

From the user-level floating-point filter, Example E-2 in Section E.4.3, "Example SIMD Floating-Point Emulation Implementation," will present only the floating-point emulation part. In order to understand the actions involved, the expected response to exceptions has to be known for all SSE/SSE2/SSE3 numeric instructions in two situations: with exceptions enabled (unmasked result), and with exceptions disabled (masked result). The latter can be found in Section 6.4, "Interrupts and Exceptions." The response to NaN operands that do not raise an exception is specified in Section 4.8.3.4, "NaNs." Operations on NaNs are explained in the same source. This response is also discussed in more detail in the next subsection, along with the unmasked and masked responses to floating-point exceptions.

E.4.2 SSE/SSE2/SSE3 Response To Floating-Point Exceptions

This subsection specifies the unmasked response expected from the SSE/SSE2/SSE3 instructions that raise floating-point exceptions. The masked response is given in parallel, as it is necessary in the emulation process of

the instructions that raise unmasked floating-point exceptions. The response to NaN operands is also included in more detail than in Section 4.8.3.4, "NaNs." For floating-point exception priority, refer to "Priority Among Simultaneous Exceptions and Interrupts" in Chapter 6, "Interrupt and Exception Handling," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.*

E.4.2.1 Numeric Exceptions

There are six classes of numeric (floating-point) exception conditions that can occur: Invalid operation (#I), Divide-by-Zero (#Z), Denormal Operand (#D), Numeric Overflow (#O), Numeric Underflow (#U), and Inexact Result (precision) (#P). #I, #Z, #D are pre-computation exceptions (floating-point faults), detected before the arithmetic operation. #O, #U, #P are post-computation exceptions (floating-point traps).

Users can control how the SSE/SSE2/SSE3 floating-point exceptions are handled by setting the mask/unmask bits in MXCSR. Masked exceptions are handled by the processor, or by software if they are combined with unmasked exceptions occurring in the same instruction. Unmasked exceptions are usually handled by the low-level exception handler, in conjunction with user-level software.

E.4.2.2 Results of Operations with NaN Operands or a NaN Result for SSE/SSE2/SSE3 Numeric Instructions

The tables below (E-1 through E-10) specify the response of SSE/SSE2/SSE3 instructions to NaN inputs, or to other inputs that lead to NaN results.

These results will be referenced by subsequent tables (e.g., E-10). Most operations do not raise an invalid exception for quiet NaN operands, but even so, they will have higher precedence over raising floating-point exceptions other than invalid operation.

Note that the single precision QNaN Indefinite value is FFC00000H, the double precision QNaN Indefinite value is FFF8000000000000H, and the Integer Indefinite value is 80000000H (not a floating-point number, but it can be the result of a conversion instruction from floating-point to integer).

For an unmasked exception, no result will be provided by the hardware to the user handler. If a user registered floating-point exception handler is invoked, it may provide a result for the excepting instruction, that will be used if execution of the application code is continued after returning from the interruption.

In Tables E-1 through Table E-12, the specified operands cause an invalid exception, unless the unmasked result is marked with "not an exception". In this latter case, the unmasked and masked results are the same.

Table E-1. ADDPS, ADDSS, SUBPS, SUBSS, MULPS, MULSS, DIVPS, DIVSS, ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, ADDSUBPS, ADDSUBPD, HADDPS, HADDPD, HSUBPS, HSUBPD

Source Operands	Masked Result	Unmasked Result
SNaN1 op[1] SNaN2	SNaN1 | 00400000H or SNaN1 | 0008000000000000H[2]	None
SNaN1 op QNaN2	SNaN1 | 00400000H or SNaN1 | 0008000000000000H[2]	None
QNaN1 op SNaN2	QNaN1	None
QNaN1 op QNaN2	QNaN1	QNaN1 (not an exception)
SNaN op real value	SNaN | 00400000H or SNaN1 | 0008000000000000H[2]	None
Real value op SNaN	SNaN | 00400000H or SNaN1 | 0008000000000000H[2]	None
QNaN op real value	QNaN	QNaN (not an exception)
Real value op QNaN	QNaN	QNaN (not an exception)

Table E-1. ADDPS, ADDSS, SUBPS, SUBSS, MULPS, MULSS, DIVPS, DIVSS, ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, ADDSUBPS, ADDSUBPD, HADDPS, HADDPD, HSUBPS, HSUBPD (Contd.)

Source Operands	Masked Result	Unmasked Result
Neither source operand is SNaN, but #I is signaled (e.g. for Inf - Inf, Inf * 0, Inf / Inf, 0/0)	Single precision or double precision QNaN Indefinite	None

NOTES:

1. For Tables E-1 to E-12: op denotes the operation to be performed.
2. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.
3. Operations involving only quiet NaNs do not raise floating-point exceptions.

Table E-2. CMPPS.EQ, CMPSS.EQ, CMPPS.ORD, CMPSS.ORD, CMPPD.EQ, CMPSD.EQ, CMPPD.ORD, CMPSD.ORD

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	00000000H or 0000000000000000H[1]	00000000H or 0000000000000000H[1] (not an exception)
Opd1 op NaN (any Opd1)	00000000H or 0000000000000000H[1]	00000000H or 0000000000000000H[1] (not an exception)

NOTE:

1. 32-bit results are for single, and 64-bit results for double precision operations.

Table E-3. CMPPS.NEQ, CMPSS.NEQ, CMPPS.UNORD, CMPSS.UNORD, CMPPD.NEQ, CMPSD.NEQ, CMPPD.UNORD, CMPSD.UNORD

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1]	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1] (not an exception)
Opd1 op NaN (any Opd1)	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1]	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1] (not an exception)

NOTE:

1. 32-bit results are for single, and 64-bit results for double precision operations.

Table E-4. CMPPS.LT, CMPSS.LT, CMPPS.LE, CMPSS.LE, CMPPD.LT, CMPSD.LT, CMPPD.LE, CMPSD.LE

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	00000000H or 0000000000000000H[1]	None
Opd1 op NaN (any Opd1)	00000000H or 0000000000000000H[1]	None

NOTE:

1. 32-bit results are for single, and 64-bit results for double precision operations.

Table E-5. CMPPS.NLT, CMPSS.NLT, CMPPS.NLE, CMPSS.NLE, CMPPD.NLT, CMPSD.NLT, CMPPD.NLE, CMPSD.NLE

Source Operands	Masked Result	Unmasked Result
NaN op Opd2 (any Opd2)	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1]	None
Opd1 op NaN (any Opd1)	FFFFFFFFH or FFFFFFFFFFFFFFFFH[1]	None

NOTE:

1. 32-bit results are for single, and 64-bit results for double precision operations.

Table E-6. COMISS, COMISD

Source Operands	Masked Result	Unmasked Result
SNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op SNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None
QNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op QNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None

Table E-7. UCOMISS, UCOMISD

Source Operands	Masked Result	Unmasked Result
SNaN op Opd2 (any Opd2)	OF, SF, AF = 000 ZF, PF, CF = 111	None
Opd1 op SNaN (any Opd1)	OF, SF, AF = 000 ZF, PF, CF = 111	None
QNaN op Opd2 (any Opd2 ≠ SNaN)	OF, SF, AF = 000 ZF, PF, CF = 111	OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)
Opd1 op QNaN (any Opd1 ≠ SNaN)	OF, SF, AF = 000 ZF, PF, CF = 111	OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)

Table E-8. CVTPS2PI, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, CVTPD2PI, CVTSD2SI, CVTTPD2PI, CVTTSD2SI, CVTPS2DQ, CVTTPS2DQ, CVTPD2DQ, CVTTPD2DQ

Source Operand	Masked Result	Unmasked Result
SNaN	80000000H or 8000000000000000[1] (Integer Indefinite)	None
QNaN	80000000H or 8000000000000000[1] (Integer Indefinite)	None

NOTE:

1. 32-bit results are for single, and 64-bit results for double precision operations.

Table E-9. MAXPS, MAXSS, MINPS, MINSS, MAXPD, MAXSD, MINPD, MINSD

Source Operands	Masked Result	Unmasked Result
Opd1 op NaN2 (any Opd1)	NaN2	None
NaN1 op Opd2 (any Opd2)	Opd2	None

NOTE:

1. SNaN and QNaN operands raise an Invalid Operation fault.

Table E-10. SQRTPS, SQRTSS, SQRTPD, SQRTSD

Source Operand	Masked Result	Unmasked Result
QNaN	QNaN	QNaN (not an exception)
SNaN	SNaN | 00400000H or SNaN | 0008000000000000H[1]	None
Source operand is not SNaN; but #I is signaled (e.g. for sqrt (-1.0))	Single precision or double precision QNaN Indefinite	None

NOTE:

1. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.

Table E-11. CVTPS2PD, CVTSS2SD

Source Operands	Masked Result	Unmasked Result
QNaN	QNaN1[1]	QNaN1[1] (not an exception)
SNaN	QNaN1[2]	None

NOTES:

1. The double precision output QNaN1 is created from the single precision input QNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by appending 29 bits equal to 0.
2. The double precision output QNaN1 is created from the single precision input SNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by pending 29 bits equal to 0. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

Table E-12. CVTPD2PS, CVTSD2SS

Source Operands	Masked Result	Unmasked Result
QNaN	QNaN1[1]	QNaN1[1] (not an exception)
SNaN	QNaN1[2]	None

NOTES:

1. The single precision output QNaN1 is created from the double precision input QNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits.
2. The single precision output QNaN1 is created from the double precision input SNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

E.4.2.3 Condition Codes, Exception Flags, and Response for Masked and Unmasked Numeric Exceptions

In the following, the masked response is what the processor provides when a masked exception is raised by an SSE/SSE2/SSE3 numeric instruction. The same response is provided by the floating-point emulator for SSE/SSE2/SSE3 numeric instructions, when certain components of the quadruple input operands generate exceptions that are masked (the emulator also generates the correct answer, as specified by IEEE Standard 754 wherever applicable, in the case when no floating-point exception occurs). The unmasked response is what the emulator provides to the user handler for those components of the packed operands of SSE/SSE2/SSE3 instructions that raise unmasked exceptions. Note that for pre-computation exceptions (floating-point faults), no result is provided to the user handler. For post-computation exceptions (floating-point traps), a result is provided to the user handler, as specified below.

In the following tables, the result is denoted by 'res', with the understanding that for the actual instruction, the destination coincides with the first source operand (except for COMISS, UCOMISS, COMISD, and UCOMISD, whose destination is the EFLAGS register).

Table E-13. #I - Invalid Operations

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSS ADDSD HADDPS HADDPD	src1 or src2[1] = SNaN	Refer to Table E-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
ADDSUBPS (the addition component) ADDSUBPD (the addition component)	src1 = +Inf, src2 = -Inf or src1 = -Inf, src2 = +Inf	res[1] = QNaN Indefinite, #IA = 1	
SUBPS SUBPD SUBSS SUBSD HSUBPS HSUBPD	src1 or src2 = SNaN	Refer to Table E-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
ADDSUBPS (the subtraction component) ADDSUBPD (the subtraction component)	src1 = +Inf, src2 = +Inf or src1 = -Inf, src2 = -Inf	res = QNaN Indefinite, #IA = 1	
MULPS MULPD	src1 or src2 = SNaN	Refer to Table E-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
MULSS MULSD	src1 = ±Inf, src2 = ±0 or src1 = ±0, src2 = ±Inf	res = QNaN Indefinite, #IA = 1	
DIVPS DIVPD	src1 or src2 = SNaN	Refer to Table E-1 for NaN operands, #IA = 1	src1, src2 unchanged; #IA = 1
DIVSS DIVSD	src1 = ±Inf, src2 = ±Inf or src1 = ±0, src2 = ±0	res = QNaN Indefinite, #IA = 1	
SQRTPS SQRTPD SQRTSS SQRTSD	src = SNaN	Refer to Table E-10 for NaN operands, #IA = 1	src unchanged, #IA = 1
	src < 0 (note that -0 < 0 is false)	res = QNaN Indefinite, #IA = 1	

Table E-13. #I - Invalid Operations (Contd.)

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
MAXPS MAXSS MAXPD MAXSD	src1 = NaN or src2 = NaN	res = src2, #IA = 1	src1, src2 unchanged; #IA = 1
MINPS MINSS MINPD MINSD	src1 = NaN or src2 = NaN	res = src2, #IA = 1	src1, src2 unchanged; #IA = 1
CMPPS.LT CMPPS.LE CMPPS.NLT CMPPS.NLE CMPSS.LT CMPSS.LE CMPSS.NLT CMPSS.NLE CMPPD.LT CMPPD.LE CMPPD.NLT CMPPD.NLE CMPSD.LT CMPSD.LE CMPSD.NLT CMPSD.NLE	src1 = NaN or src2 = NaN	Refer to Table E-4 and Table E-5 for NaN operands; #IA = 1	src1, src2 unchanged; #IA = 1
COMISS COMISD	src1 = NaN or src2 = NaN	Refer to Table E-6 for NaN operands	src1, src2, EFLAGS unchanged; #IA = 1
UCOMISS UCOMISD	src1 = SNaN or src2 = SNaN	Refer to Table E-7 for NaN operands	src1, src2, EFLAGS unchanged; #IA = 1
CVTPS2PI CVTSS2SI CVTPD2PI CVTSD2SI CVTPS2DQ CVTPD2DQ	src = NaN, ±Inf, or $\|(src)_{rnd}\| > 7FFFFFFFH$ and $(src)_{rnd} \neq 80000000H$ See Note[2] for information on rnd.	res = Integer Indefinite, #IA = 1	src unchanged, #IA = 1
CVTTPS2PI CVTTSS2SI CVTTPD2PI CVTTSD2SI CVTTPS2DQ CVTTPD2DQ	src = NaN, ±Inf, or $\|(src)_{rz}\| > 7FFFFFFFH$ and $(src)_{rz} \neq 80000000H$ See Note[2] for information on rz.	res = Integer Indefinite, #IA = 1	src unchanged, #IA = 1

Table E-13. #I - Invalid Operations (Contd.)

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
CVTPS2PD CVTSS2SD	src = SNAN	Refer to Table E-11 for NaN operands	src unchanged, #IA = 1
CVTPD2PS CVTSD2SS	src = SNAN	Refer to Table E-12 for NaN operands	src unchanged, #IA = 1

NOTES:

1. For Tables E-13 to E-18:
   - src denotes the single source operand of a unary operation.
   - src1, src2 denote the first and second source operand of a binary operation.
   - res denotes the numerical result of an operation.
2. rnd signifies the user rounding mode from MXCSR, and rz signifies the rounding mode toward zero. (truncate), when rounding a floating-point value to an integer. For more information, refer to Table 4-8.
3. For NAN encodings, see Table 4-3.

Table E-14. #Z - Divide-by-Zero

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
DIVPS DIVSS DIVPD DIVPS	src1 = finite non-zero (normal, or denormal) src2 = ±0	res = ±Inf, #ZE = 1	src1, src2 unchanged; #ZE = 1

Table E-15. #D - Denormal Operand

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD MAXPS MAXPD MINPS MINPD ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD MAXSS MAXSD MINSS MINSD CVTPS2PD CVTSS2SD CVTPD2PS CVTSD2SS	src1 = denormal[1] or src2 = denormal (and the DAZ bit in MXCSR is 0)	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked post-computation exception occurs; #DE = 1.	src1, src2 unchanged; #DE = 1 Note that SQRT, CVTPS2PD, CVTSS2SD, CVTPD2PS, CVTSD2SS have only 1 src.
CMPPS CMPPD CMPSS CMPSD	src1 = denormal[1] or src2 = denormal (and the DAZ bit in MXCSR is 0)	Comparison result, stored in the destination register; #DE = 1	src1, src2 unchanged; #DE = 1
COMISS COMISD UCOMISS UCOMISD	src1 = denormal[1] or src2 = denormal (and the DAZ bit in MXCSR is 0)	Comparison result, stored in the EFLAGS register; #DE = 1	src1, src2 unchanged; #DE = 1

NOTE:

1. For denormal encodings, see Section 4.8.3.2, “Normalized and Denormalized Finite Numbers.”

Table E-16. #O - Numeric Overflow

Instruction	Condition	Masked Response			Unmasked Response and Exception Code
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	Rounded result > largest single precision finite normal value	**Rounding**	**Sign**	**Result & Status Flags**	res = (result calculated with unbounded exponent and rounded to the destination precision) / 2^{192} #OE = 1 #PE = 1 if the result is inexact

		To nearest	+
			-
				#OE = 1, #PE = 1
				res = $+\infty$
				res = $-\infty$	

		Toward $-\infty$	+
			-
				#OE = 1, #PE = 1
				res = 1.11...1 * 2^{127}
				res = $-\infty$	

		Toward $+\infty$	+
			-
				#OE = 1, #PE = 1
				res = $+\infty$
				res = -1.11...1 * 2^{127}	

		Toward 0	+
			-
				#OE = 1, #PE = 1
				res = 1.11...1 * 2^{127}
				res = -1.11...1 * 2^{127}	

ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	Rounded result > largest double precision finite normal value	**Rounding**	**Sign**	**Result & Status Flags**	res = (result calculated with unbounded exponent and rounded to the destination precision) / 2^{1536} ▪ #OE = 1 ▪ #PE = 1 if the result is inexact

		To nearest	+
			-
				#OE = 1, #PE = 1
				res = $+\infty$
				res = $-\infty$	

		Toward $-\infty$	+
			-
				#OE = 1, #PE = 1
				res = 1.11...1 * 2^{1023}
				res = $-\infty$	

		Toward $+\infty$	+
			-
				#OE = 1, #PE = 1
				res = $+\infty$
				res = -1.11...1 * 2^{1023}	

		Toward 0	+
			-
				#OE = 1, #PE = 1
				res = 1.11...1 * 2^{1023}
				res = -1.11...1 * 2^{1023}	

Table E-17. #U - Numeric Underflow

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	Result calculated with unbounded exponent and rounded to the destination precision < smallest single precision finite normal value.	res = ±0, denormal, or normal #UE = 1 and #PE = 1, but only if the result is inexact	res = (result calculated with unbounded exponent and rounded to the destination precision) * 2^{192} ▪ #UE = 1 ▪ #PE = 1 if the result is inexact
ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	Result calculated with unbounded exponent and rounded to the destination precision < smallest double precision finite normal value.	res = ±0, denormal or normal #UE = 1 and #PE = 1, but only if the result is inexact	res = (result calculated with unbounded exponent and rounded to the destination precision) * 2^{1536} ▪ #UE = 1 ▪ #PE = 1 if the result is inexact

Table E-18. #P - Inexact Result (Precision)

Instruction	Condition	Masked Response	Unmasked Response and Exception Code
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD CVTDQ2PS CVTPI2PS CVTPS2PI CVTPS2DQ CVTPD2PI CVTPD2DQ CVTPD2PS CVTTPS2PI CVTTPD2PI CVTTPD2DQ CVTTPS2DQ ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD CVTSI2SS CVTSS2SI CVTSD2SI CVTSD2SS CVTTSS2SI CVTTSD2SI	The result is not exactly representable in the destination format.	res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked underflow or overflow conditions occur (this exception can occur in the presence of a masked underflow or overflow); #PE = 1.	Only if no underflow/overflow condition occurred, or if the corresponding exceptions are masked: ▪ Set #OE if masked overflow and set result as described above for masked overflow. ▪ Set #UE if masked underflow and set result as described above for masked underflow. If neither underflow nor overflow, res equals the result rounded to the destination precision and using the bounded exponent set #PE = 1.

E.4.3 Example SIMD Floating-Point Emulation Implementation

The sample code listed below may be considered as being part of a user-level floating-point exception filter for the SSE/SSE2/SSE3 numeric instructions. It is assumed that the filter function is invoked by a low-level exception handler (invoked for exception 19 when an unmasked floating-point exception occurs), and that it operates as explained in Section E.4.1, "Floating-Point Emulation." The sample code does the emulation only for the SSE instructions for addition, subtraction, multiplication, and division. For this, it uses C code and x87 FPU operations. Operations corresponding to other SSE/SSE2/SSE3 numeric instructions can be emulated similarly. The example assumes that the emulation function receives a pointer to a data structure specifying a number of input parameters: the operation that caused the exception, a set of sub-operands (unpacked, of type float), the rounding mode

(the precision is always single), exception masks (having the same relative bit positions as in the MXCSR but starting from bit 0 in an unsigned integer), and flush-to-zero and denormals-are-zeros indicators.

The output parameters are a floating-point result (of type float), the cause of the exception (identified by constants not explicitly defined below), and the exception status flags. The corresponding C definition is:

```
typedef struct {
  unsigned int operation;              //SSE or SSE2 operation: ADDPS, ADDSS, ...
  unsigned int operand1_uint32; //first operand value
  unsigned int operand2_uint32; //second operand value (if any)
  float result_fval; // result value (if any)
  unsigned int rounding_mode; //rounding mode
  unsigned int exc_masks; //exception masks, in the order P,U,O,Z,D,I
  unsigned int exception_cause; //exception cause
  unsigned int status_flag_inexact; //inexact status flag
  unsigned int status_flag_underflow; //underflow status flag
  unsigned int status_flag_overflow; //overflow status flag
  unsigned int status_flag_divide_by_zero;
                                                //divide by zero status flag
  unsigned int status_flag_denormal_operand;
                                                //denormal operand status flag
  unsigned int status_flag_invalid_operation;
                                                //invalid operation status flag
  unsigned int ftz; // flush-to-zero flag
  unsigned int daz; // denormals-are-zeros flag
} EXC_ENV;
```

The arithmetic operations exemplified are emulated as follows:

1. If the denormals-are-zeros mode is enabled (the DAZ bit in MXCSR is set to 1), replace all the denormal inputs with zeroes of the same sign (the denormal flag is not affected by this change).
2. Perform the operation using x87 FPU instructions, with exceptions disabled, the original user rounding mode, and single precision. This reveals invalid, denormal, or divide-by-zero exceptions (if there are any) and stores the result in memory as a double precision value (whose exponent range is large enough to look like "unbounded" to the result of the single precision computation).
3. If no unmasked exceptions were detected, determine if the magnitude of the result is less than the smallest normal number that can be represented in single precision format, or greater than the largest normal number that can be represented in single precision format (huge). If an unmasked overflow or underflow occurs, calculate the scaled result that will be handed to the user exception handler, as specified by IEEE Standard 754.
4. If no exception was raised, calculate the result with a "bounded" exponent. If the result is tiny, it requires denormalization (shifting the significand right while incrementing the exponent to bring it into the admissible range of [-126,+127] for single precision floating-point numbers).

   The result obtained in step 2 cannot be used because it might incur a double rounding error (it was rounded to 24 bits in step 2, and might have to be rounded again in the denormalization process). To overcome this is, calculate the result as a double precision value, and store it to memory in single precision format.

   Rounding first to 53 bits in the significand, and then to 24 never causes a double rounding error (exact properties exist that state when double-rounding error occurs, but for the elementary arithmetic operations, the rule of thumb is that if an infinitely precise result is rounded to 2p+1 bits and then again to p bits, the result is the same as when rounding directly to p bits, which means that no double-rounding error occurs).
5. If the result is inexact and the inexact exceptions are unmasked, the calculated result will be delivered to the user floating-point exception handler.
6. The flush-to-zero case is dealt with if the result is tiny.

7. The emulation function returns RAISE_EXCEPTION to the filter function if an exception has to be raised (the exception_cause field indicates the cause). Otherwise, the emulation function returns DO_NOT_ RAISE_EXCEPTION. In the first case, the result is provided by the user exception handler called by the filter function. In the second case, it is provided by the emulation function. The filter function has to collect all the partial results, and to assemble the scalar or packed result that is used if execution is to continue.

Example E-2. SIMD Floating-Point Emulation

```
// masks for individual status word bits
#define PRECISION_MASK 20H
#define UNDERFLOW_MASK 10H
#define OVERFLOW_MASK 08H
#define ZERODIVIDE_MASK 04H
#define DENORMAL_MASK 02H
#define INVALID_MASK 01H

// 32-bit constants
static unsigned ZEROF_ARRAY[] = {00000000H};
#define  ZEROF *(float *) ZEROF_ARRAY
    // +0.0
static unsigned NZEROF_ARRAY[] = {80000000H};
#define  NZEROF *(float *) NZEROF_ARRAY
    // -0.0
static unsigned POSINFF_ARRAY[] = {7f800000H};
#define POSINFF *(float *)POSINFF_ARRAY
    // +Inf
static unsigned NEGINFF_ARRAY[] = {ff800000H};
#define NEGINFF *(float *)NEGINFF_ARRAY
    // -Inf

// 64-bit constants
static unsigned MIN_SINGLE_NORMAL_ARRAY [] = {00000000H, 38100000H};
#define MIN_SINGLE_NORMAL *(double *)MIN_SINGLE_NORMAL_ARRAY
    // +1.0 * 2^-126
static unsigned MAX_SINGLE_NORMAL_ARRAY [] = {70000000H, 47effffH};
#define MAX_SINGLE_NORMAL *(double *)MAX_SINGLE_NORMAL_ARRAY
    // +1.1...1*2^127
static unsigned TWO_TO_192_ARRAY[] = {00000000H, 4bf00000H};
#define TWO_TO_192 *(double *)TWO_TO_192_ARRAY
    // +1.0 * 2^192
static unsigned TWO_TO_M192_ARRAY[] = {00000000H, 33f00000H};
#define TWO_TO_M192 *(double *)TWO_TO_M192_ARRAY
    // +1.0 * 2^-192

// auxiliary functions
static int isnanf (unsigned int ); // returns 1 if f is a NaN, and 0 otherwise
static float quietf (unsigned int ); // converts a signaling NaN to a quiet
                          // NaN, and leaves a quiet NaN unchanged
static unsigned int check_for_daz (unsigned int ); // converts denormals
                          // to zeros of the same sign;
                          // does not affect any status flags

// emulation of SSE and SSE2 instructions using
// C code and x87 FPU instructions

unsigned int
simd_fp_emulate (EXC_ENV *exc_env)

{

  int uiopd1; // first operand of the add, subtract, multiply, or divide
  int uiopd2; // second operand of the add, subtract, multiply, or divide
  float res; // result of the add, subtract, multiply, or divide
  double dbl_res24; // result with 24-bit significand, but "unbounded" exponent
```

```
    // (needed to check tininess, to provide a scaled result to
    // an underflow/overflow trap handler, and in flush-to-zero mode)
double dbl_res;  // result in double precision format (needed to avoid a
   // double rounding error when denormalizing)
unsigned int result_tiny;
unsigned int result_huge;
unsigned short int sw; // 16 bits
unsigned short int cw; // 16 bits


// have to check first for faults (V, D, Z), and then for traps (O, U, I)

// initialize x87 FPU (floating-point exceptions are masked)
_asm {
  fninit;
}

result_tiny = 0;
result_huge = 0;

switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
  case SUBPS:
  case SUBSS:
  case MULPS:
  case MULSS:
  case DIVPS:
  case DIVSS:

    uiopd1 = exc_env->operand1_uint32; // copy as unsigned int
     // do not copy as float to avoid conversion
     // of SNaN to QNaN by compiled code
    uiopd2 = exc_env->operand2_uint32;
     // do not copy as float to avoid conversion of SNaN
     // to QNaN by compiled code
 uiopd1 = check_for_daz (uiopd1); // operand1 = +0.0 * operand1 if it is
    // denormal and DAZ=1
    uiopd2 = check_for_daz (uiopd2); // operand2 = +0.0 * operand2 if it is
     // denormal and DAZ=1

    // execute the operation and check whether the invalid, denormal, or
    // divide by zero flags are set and the respective exceptions enabled

    // set control word with rounding mode set to exc_env->rounding_mode,
    // single precision, and all exceptions disabled
    switch (exc_env->rounding_mode) {
      case ROUND_TO_NEAREST:
        cw = 003fH; // round to nearest, single precision, exceptions masked
        break;
      case ROUND_DOWN:
        cw = 043fH; // round down, single precision, exceptions masked
        break;
      case ROUND_UP:
        cw = 083fH; // round up, single precision, exceptions masked
        break;
      case ROUND_TO_ZERO:
        cw = 0c3fH; // round to zero, single precision, exceptions masked
        break;
      default:
        ;
    }
    __asm {
      fldcw WORD PTR cw;
```

```
}

// compute result and round to the destination precision, with
// "unbounded" exponent (first IEEE rounding)
switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
    // perform the addition
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      faddp st(1), st(0); // may set inexact or invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case SUBPS:
  case SUBSS:
    // perform the subtraction
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fsubp st(1), st(0); // may set the inexact or invalid status flags

     // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case MULPS:
  case MULSS:
    // perform the multiplication
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fmulp st(1), st(0); // may set inexact or invalid status flags

     // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case DIVPS:
  case DIVSS:
    // perform the division
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fdivp st(1), st(0); // may set the inexact, divide by zero, or
                          // invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;
```

```
        default:
          ; // will never occur

      }

      // read status word
      __asm {
        fstsw WORD PTR sw;
}

    if (sw & ZERODIVIDE_MASK)
    sw = sw & ~DENORMAL_MASK; // clear D flag for (denormal / 0)

      // if invalid flag is set, and invalid exceptions are enabled, take trap
      if (!(exc_env->exc_masks & INVALID_MASK) && (sw & INVALID_MASK)) {
        exc_env->status_flag_invalid_operation = 1;
        exc_env->exception_cause = INVALID_OPERATION;
        return (RAISE_EXCEPTION);
      }

    // checking for NaN operands has priority over denormal exceptions;
    // also fix for the SSE and SSE2
    // differences in treating two NaN inputs between the
    // instructions and other IA-32 instructions
    if (isnanf (uiopd1) || isnanf (uiopd2)) {

        if (isnanf (uiopd1) && isnanf (uiopd2))
            exc_env->result_fval = quietf (uiopd1);
        else
            exc_env->result_fval = (float)dbl_res24; // exact

        if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
        return (DO_NOT_RAISE_EXCEPTION);
      }

      // if denormal flag set, and denormal exceptions are enabled, take trap
      if (!(exc_env->exc_masks & DENORMAL_MASK) && (sw & DENORMAL_MASK)) {
        exc_env->status_flag_denormal_operand = 1;
        exc_env->exception_cause = DENORMAL_OPERAND;
        return (RAISE_EXCEPTION);
      }

      // if divide by zero flag set, and divide by zero exceptions are
      // enabled, take trap (for divide only)
      if (!(exc_env->exc_masks & ZERODIVIDE_MASK) && (sw & ZERODIVIDE_MASK)) {
        exc_env->status_flag_divide_by_zero = 1;
        exc_env->exception_cause = DIVIDE_BY_ZERO;
        return (RAISE_EXCEPTION);
      }

      // done if the result is a NaN (QNaN Indefinite)
      res = (float)dbl_res24;
      if (isnanf (*(unsigned int *)&res)) {
        exc_env->result_fval = res; // exact
        exc_env->status_flag_invalid_operation = 1;
        return (DO_NOT_RAISE_EXCEPTION);
      }

      // dbl_res24 is not a NaN at this point

      if (sw & DENORMAL_MASK) exc_env->status_flag_denormal_operand = 1;

      // Note: (dbl_res24 == 0.0 && sw & PRECISION_MASK) cannot occur
      if (-MIN_SINGLE_NORMAL < dbl_res24 && dbl_res24 < 0.0 ||
            0.0 < dbl_res24 && dbl_res24 < MIN_SINGLE_NORMAL) {
```

```
  result_tiny = 1;
}

// check if the result is huge
if (NEGINFF < dbl_res24 && dbl_res24 < -MAX_SINGLE_NORMAL ||
    MAX_SINGLE_NORMAL < dbl_res24 && dbl_res24 < POSINFF) {
  result_huge = 1;
}

// at this point, there are no enabled I,D, or Z exceptions
// to take; the instr.
// might lead to an enabled underflow, enabled underflow and inexact,
// enabled overflow, enabled overflow and inexact, enabled inexact, or
// none of these; if there are no U or O enabled exceptions, re-execute
// the instruction using IA-32 double precision format, and the
// user's rounding mode; exceptions must have
// been disabled before calling
// this function; an inexact exception may be reported on the 53-bit
// fsubp, fmulp, or on both the 53-bit and 24-bit conversions, while an
// overflow or underflow (with traps disabled) may be reported on the
// conversion from dbl_res to res

// check whether there is an underflow, overflow,
// or inexact trap to be taken
// if the underflow traps are enabled and the result is
// tiny, take underflow trap

if (!(exc_env->exc_masks & UNDERFLOW_MASK) && result_tiny) {
  dbl_res24 = TWO_TO_192 * dbl_res24; // exact
  exc_env->status_flag_underflow = 1;
  exc_env->exception_cause = UNDERFLOW;
  exc_env->result_fval = (float)dbl_res24; // exact
  if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
  return (RAISE_EXCEPTION);
}

// if overflow traps are enabled and the result is huge, take
// overflow trap
if (!(exc_env->exc_masks & OVERFLOW_MASK) &&  result_huge) {
  dbl_res24 = TWO_TO_M192 * dbl_res24; // exact
  exc_env->status_flag_overflow = 1;
  exc_env->exception_cause = OVERFLOW;
  exc_env->result_fval = (float)dbl_res24; // exact
  if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
  return (RAISE_EXCEPTION);
}

// set control word with rounding mode set to exc_env->rounding_mode,
// double precision, and all exceptions disabled
cw = cw | 0200H; // set precision to double
__asm {
  fldcw WORD PTR cw;
}

switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
    // perform the addition
    __asm {
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal status flag
      fld DWORD PTR uiopd2; // may set the denormal status flag
      faddp st(1), st(0); // rounded to 53 bits, may set the inexact
                          // status flag
```

```
        // store result
        fstp  QWORD PTR dbl_res; // exact, will not set any flag
      }
      break;

    case SUBPS:
    case SUBSS:
      // perform the subtraction
      __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fsubp st(1), st(0);   // rounded to 53 bits, may set the inexact
                              // status flag
        // store result
        fstp  QWORD PTR dbl_res; // exact, will not set any flag
      }
      break;

    case MULPS:
    case MULSS:
      // perform the multiplication
      __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fmulp st(1), st(0);   // rounded to 53 bits, exact
       // store result
        fstp  QWORD PTR dbl_res; // exact, will not set any flag
      }
      break;

    case DIVPS:
    case DIVSS:
      // perform the division
      __asm {
        // load input operands
        fld DWORD PTR uiopd1; // may set the denormal status flag
        fld DWORD PTR uiopd2; // may set the denormal status flag
        fdivp st(1), st(0);   // rounded to 53 bits, may set the inexact
                              // status flag
        // store result
        fstp  QWORD PTR dbl_res; // exact, will not set any flag
      }
      break;

    default:
      ; // will never occur

  }

  // calculate result for the case an inexact trap has to be taken, or
  // when no trap occurs (second IEEE rounding)
  res = (float)dbl_res;
      // may set P, U or O; may also involve denormalizing the result

  // read status word
  __asm {
    fstsw WORD PTR sw;
  }

  // if inexact traps are enabled and result is inexact, take inexact trap
  if (!(exc_env->exc_masks & PRECISION_MASK) &&
      ((sw & PRECISION_MASK) || (exc_env->ftz && result_tiny))) {
    exc_env->status_flag_inexact = 1;
```

```
    exc_env->exception_cause = INEXACT;
    if (result_tiny) {
      exc_env->status_flag_underflow = 1;

      // if ftz = 1 and result is tiny, result = 0.0
      // (no need to check for underflow traps disabled: result tiny and
      // underflow traps enabled would have caused taking an underflow
      // trap above)
      if (exc_env->ftz) {
        if (res > 0.0)
          res = ZEROF;
        else if (res < 0.0)
          res = NZEROF;
        // else leave res unchanged
      }
    }
    if (result_huge) exc_env->status_flag_overflow = 1;
    exc_env->result_fval = res;
    return (RAISE_EXCEPTION);
  }

  // if it got here, then there is no trap to be taken; the following must
  // hold: ((the MXCSR U exceptions are disabled  or
  //
  // the MXCSR underflow exceptions are enabled and the underflow flag is
  // clear and (the inexact flag is set or the inexact flag is clear and
  // the 24-bit result with unbounded exponent is not tiny)))
  // and (the MXCSR overflow traps are disabled or the overflow flag is
  // clear) and (the MXCSR inexact traps are disabled or the inexact flag
  // is clear)
  //
  // in this case, the result has to be delivered (the status flags are
  // sticky, so they are all set correctly already)

  // read status word to see if result is inexact
  __asm {
    fstsw WORD PTR sw;
  }

  if (sw & UNDERFLOW_MASK) exc_env->status_flag_underflow = 1;
  if (sw & OVERFLOW_MASK) exc_env->status_flag_overflow = 1;
  if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;

  // if ftz = 1, and result is tiny (underflow traps must be disabled),
  // result = 0.0
  if (exc_env->ftz && result_tiny) {
    if (res > 0.0)
      res = ZEROF;
    else if (res < 0.0)
      res = NZEROF;
    // else leave res unchanged

    exc_env->status_flag_inexact = 1;
    exc_env->status_flag_underflow = 1;
  }

  exc_env->result_fval = res;
  if (sw & ZERODIVIDE_MASK) exc_env->status_flag_divide_by_zero = 1;
  if (sw & DENORMAL_MASK) exc_env->status_flag_denormal= 1;
  if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
  return (DO_NOT_RAISE_EXCEPTION);

  break;

case CMPPS:
```

```
    case CMPSS:

      ...

      break;

    case COMISS:
    case UCOMISS:

      ...

      break;

    case CVTPI2PS:
    case CVTSI2SS:

      ...

      break;

    case CVTPS2PI:
    case CVTSS2SI:
    case CVTTPS2PI:
    case CVTTSS2SI:

      ...

      break;

    case MAXPS:
    case MAXSS:
    case MINPS:
    case MINSS:

      ...

      break;

    case SQRTPS:
    case SQRTSS:

      ...

      break;

...

case UNSPEC:

      ...

      break;

    default:
      ...

  }

}
```

9. Updates to Chapter 2, Volume 2A

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

Changes to this chapter: Correction to Effective Address ECX/CX/CL/MM1/XMM1 row of Table 2-1 "16-Bit Addressing Forms with the ModR/M Byte". Minor update to Section 2.6.1 "Instruction Format and EVEX". Corrections in Table 2-43 "EVEX Instructions in each Exception Class ".

CHAPTER 2
INSTRUCTION FORMAT

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).

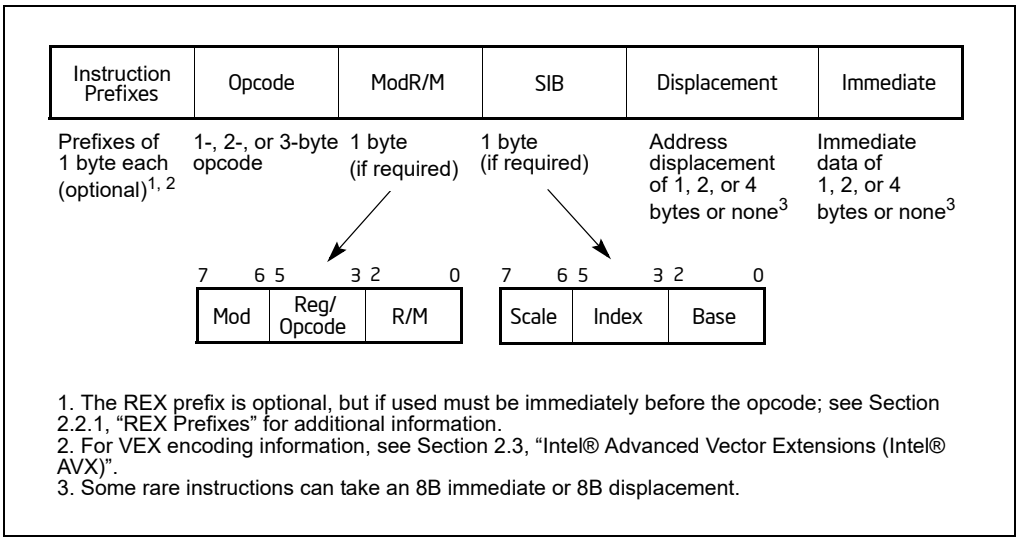


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

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/REPNZ 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.

  - 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 hints[1]:
    - 2EH—Branch not taken (used only with J*cc* instructions).
    - 3EH—Branch taken (used only with J*cc* 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 (J*cc*). 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, CVTDQ2PD 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.

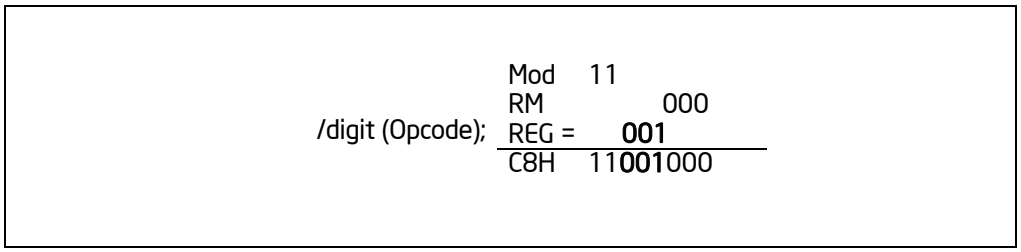


Figure 2-2. Table Interpretation of ModR/M Byte (C8H)

Table 2-1. 16-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 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 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 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)**							
[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:

1. 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.

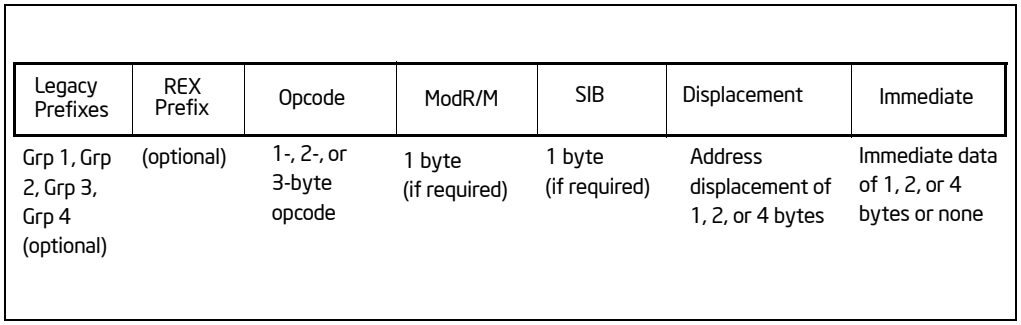


Figure 2-3. Prefix Ordering in 64-bit Mode

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 though 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

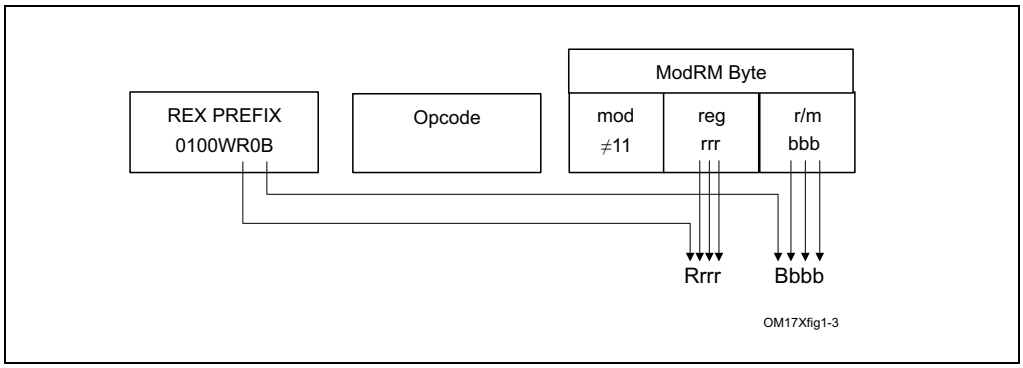


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

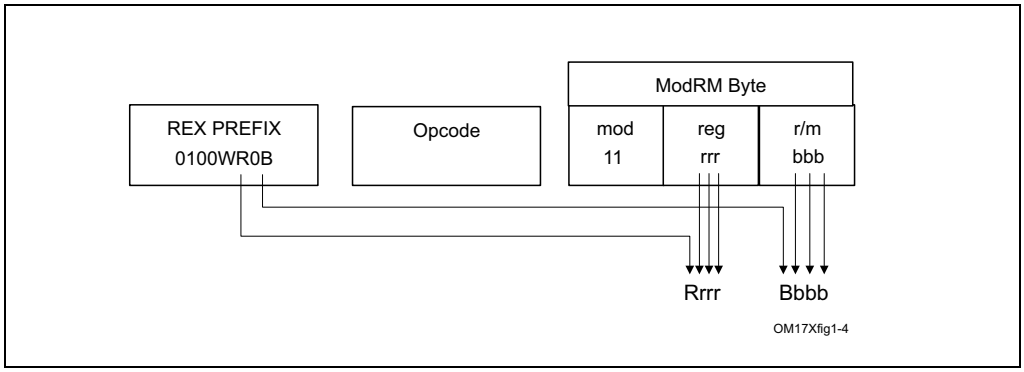


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

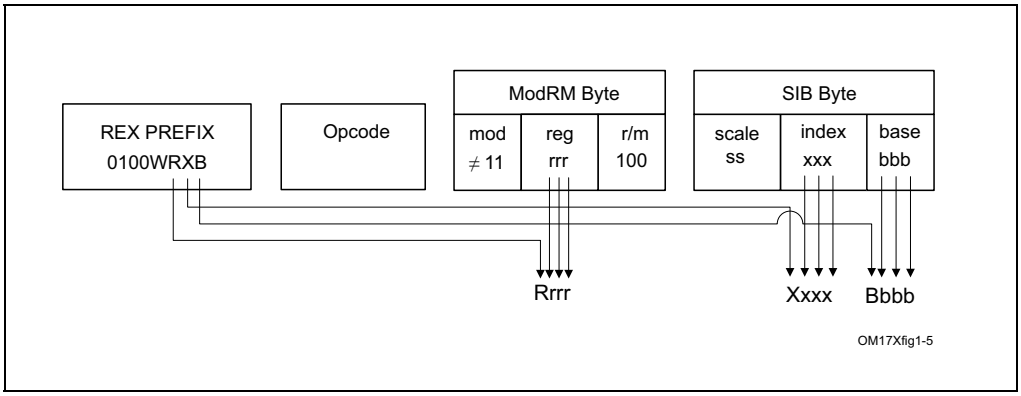


Figure 2-6. Memory Addressing With a SIB Byte

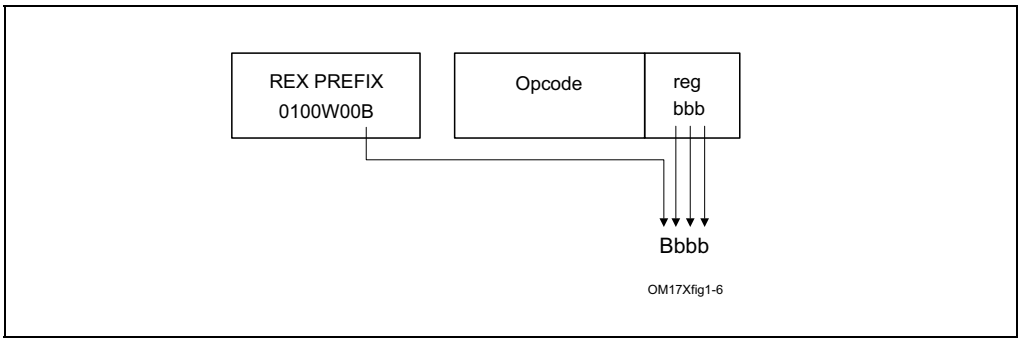


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

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 ±2GB 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	Must use SIB form with normal (zero-based) displacement addressing
	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.

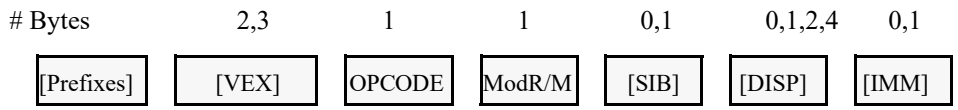


Figure 2-8. Instruction Encoding Format with VEX Prefix

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 proceeding VEX will #UD.

2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
- Vector length encoding: This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  - — Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
  - — Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  - — Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

  Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.
- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

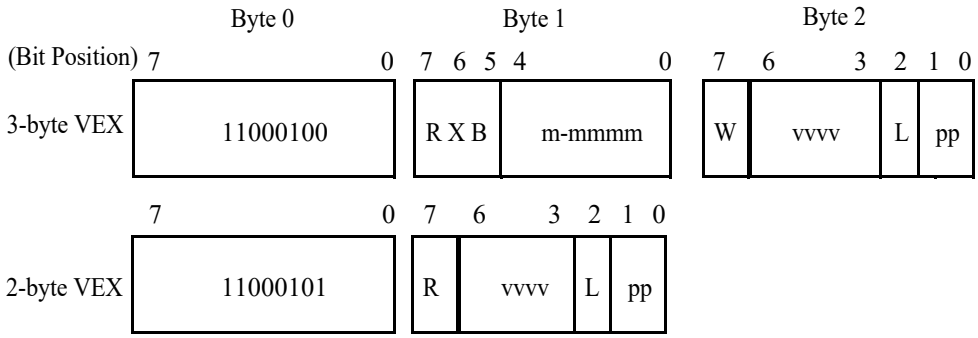
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form
1: Same as REX.R=0 (must be 1 in 32-bit mode)
0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form
1: Same as REX.X=0 (must be 1 in 32-bit mode)
0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form
1: Same as REX.B=0 (Ignored in 32-bit mode).
0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

m-mmmm:
00000: Reserved for future use (will #UD)
00001: implied 0F leading opcode byte
00010: implied 0F 38 leading opcode bytes
00011: implied 0F 3A leading opcode bytes
00100-11111: Reserved for future use (will #UD)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length
0: scalar or 128-bit vector
1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix
00: None
01: 66
10: F3
11: F2

Figure 2-9. VEX 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.NDD.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

2.3.6.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm"

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

Table 2-10. VEX.m-mmmm interpretation

VEX.m-mmmm	Implied Leading Opcode Bytes
00000B	Reserved
00001B	0F
00010B	0F 38
00011B	0F 3A
00100-11111B	Reserved
(2-byte VEX)	0F

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L"

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If "VEX.L = 1", it indicates 256-bit vector operation. "VEX.L = 0" indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero's bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

Table 2-11. VEX.L interpretation

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

2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp"

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

Table 2-12. VEX.pp interpretation

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

2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

2.3.8 The MODRM, SIB, and Displacement Bytes

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

2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

2.3.10 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

2.3.10.1 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 R8L-R15L 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, R8L-R15L 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 (In decimal) Base = (In binary) Base =				EAX/ R8L 0 000	ECX/ R9L 1 001	EDX/ R10L 2 010	EBX/ R11L 3 011	ESP/ R12L 4 100	EBP/ R13L[1] 5 101	ESI/ R14L 6 110	EDI/ R15L 7 111
Scaled Index		**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*1	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
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*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

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte (Contd.)

VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*4	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 89 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
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*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:

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)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINSD, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, VBLENDVPD, VBLENDVPS, (V)LDDQU***, (V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB, (V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRSW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ, VPERMILPD, VPERMILPS, 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)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128
Type 7	(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	VZEROALL, VZEROUPPER
Type 11	VCVTPH2PS, VCVTPS2PH
Type 12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

(*) - Additional exception restrictions are present - see the Instruction description for details

(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.

(***) - PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM 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, VPBLENDD, 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, VPMULHRSW, VPMULHW/LW, VPMULLD, VPMULUDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR	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, VPMOVMSKB, 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

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

Table 2-19. 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.4.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-20. 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 Bytes or less 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.[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. PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM and LDDQU 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.4.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

Table 2-23. 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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

Table 2-24. 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.4.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-25. 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 XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ≠ 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.4.9 Exception Type 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 XCR0[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	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 Exception 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 XCR0[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'.
	X	X	X	X	If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.
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.

2.5 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 Exception Conditions for 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 GRP 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
See Table 2-29	ANDN, 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. Exception Definition (VEX-Encoded GPR Instructions)

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	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	If alignment checking is enabled and an unaligned memory reference 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.

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.

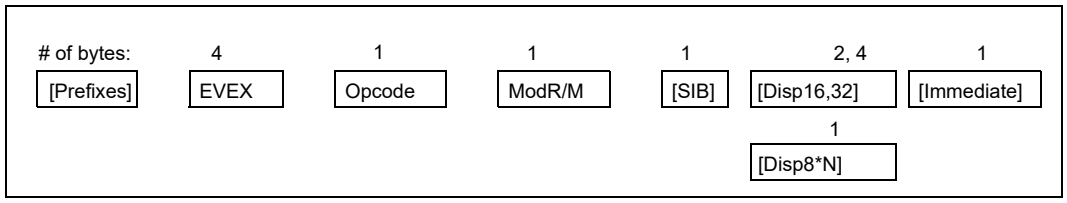


Figure 2-10. AVX-512 Instruction Format and the EVEX Prefix

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).

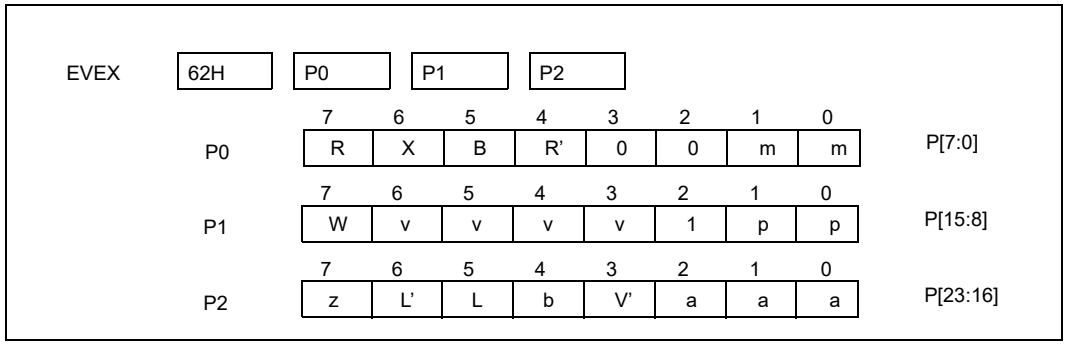


Figure 2-11. Bit Field Layout of the EVEX Prefix

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).
EVEXR'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg.
EVEXX	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.vvvv	NDS register specifier	P[14 : 11]	Same as VEX.vvvv.
EVEXV'	High-16 NDS/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present.
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
NDS/NDD	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource 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
NDS/NDD	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
NDS	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 then 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	1Tuple
	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	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
FP Instructions w/o rounding semantic, can cause #XF	SAE control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Load+op Instructions w/ memory source	Broadcast Control		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 of 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	xx1xxx11b

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)	
EVEXR'	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
EVEXV'	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 = 0xxxb	None

NOTES:

1. E.g., VBROADCASTMxxx, 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 K0.

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 #XF.	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	Half Vector Length, w/ fault suppression	Yes
Type E11NF	VCVTPS2PH	Half Vector Length, no 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, VCVTPS2DQ, VCVTTPD2DQ, VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDxxxPD, VFMADDxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS
Type E3	VADDSD, VADDSS, VCMPSD, VCMPSS, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTTSD2SI, VCVTTSS2SI, VUCOMISD, VUCOMISS VCVTSD2USI, VCVTTSD2USI, VCVTSS2USI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VXORPD, VXORPS, VPSLLVD, VPSLLVQ, VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VFPCLASSPD, VFPCLASSPS, VPCMPD, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[1], VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPSRAVW, VPSRAVD, VPSRAVW, VPSRAVQ, VPMADD52LUQ, VPMADD52HUQ
E4.nb[2]	VMOVUPD, VMOVUPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VPCMPB, VPCMPW, VPCMPUB, VPCMPUW, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VCOMPRESSPD, VCOMPRESSPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW, VPSLLW, VPSRAW, VPSRLW, VPSLLVW, VPSRLVW
Type E4NF	VPACKSSDW, VPACKUSDW VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ, VALIGND, VALIGNQ, VPCONFLICTD, VPCONFLICTQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFI64X2, VSHUFF32X4, VSHUFF64X2, VPMULTISHIFTQB
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	VCVTDQ2PD, PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ VCVTUDQ2PD
Type E5NF	VMOVDDUP
Type E6	VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VFPCLASSSD, VFPCLASSSS, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E6NF	VEXTRACTF32X4, VEXTRACTF64X2, VEXTRACTF32X8, VINSERTF32X4, VINSERTF64X2, VINSERTF64X4, VINSERTF32X8, VINSERTI32X4, VINSERTI64X2, VINSERTI64X4, VINSERTI32X8, VEXTRACTI32X4, VEXTRACTI64X2, VEXTRACTI32X8, VEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D
Type E7NM.128[4]	VMOVLHPS, VMOVHLPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)[5], VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVB2M, VPMOVD2M, VPMOVQ2M, VPMOVW2M

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
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	(VCVTSI2SD, 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	VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, 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: ▪ 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: ▪ 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: ▪ 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	If EVEX.B=1, alignment checking is enabled, and an unaligned memory reference of 8 bytes or less 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: ▪ 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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.OSX-MMEXCPT[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: ▪ 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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	If EVEX.B=1, alignment checking is enabled, and an unaligned memory reference of 8 bytes or less 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: ▪ 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: ▪ 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		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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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		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 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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 (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	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: ▪ 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 (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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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 (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 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	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7.9 Exceptions Type E10

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: ▪ 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less 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: ▪ 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		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: ▪ 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: ▪ 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.
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.

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: ▪ 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: ▪ 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	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

10. Updates to Chapter 3, Volume 2A

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

Changes to this chapter: Define "m80bcd" in Section 3.1.1.3. Updated Figure 3-2 "Memory Bit Indexing" with missing numbers.

Updates to the following instructions are covered here with change bars: ADC, ADD, AND, BNDLDX, BNDSTX, BSF, BSR, CMP, CPUID, and FBLD.

  - — **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.
  - — **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.
  - — **W0:** EVEX.W=0.
  - — **W1:** EVEX.W=1.
  - — **WIG:** EVEX.W bit ignored
- **opcode** — Instruction opcode.
- In general, the encoding of EVEX.R and R’, EVEX.X and X’, and EVEX.B and B’ fields are not shown explicitly in the opcode column.

3.1.1.3 Instruction Column in the Opcode Summary Table

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

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

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **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 ($i \leftarrow 0$ through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.

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

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

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

- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.
- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — the destination in an instruction. This field is encoded by reg_field.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

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

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tupletypes. The tupletype 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.

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).

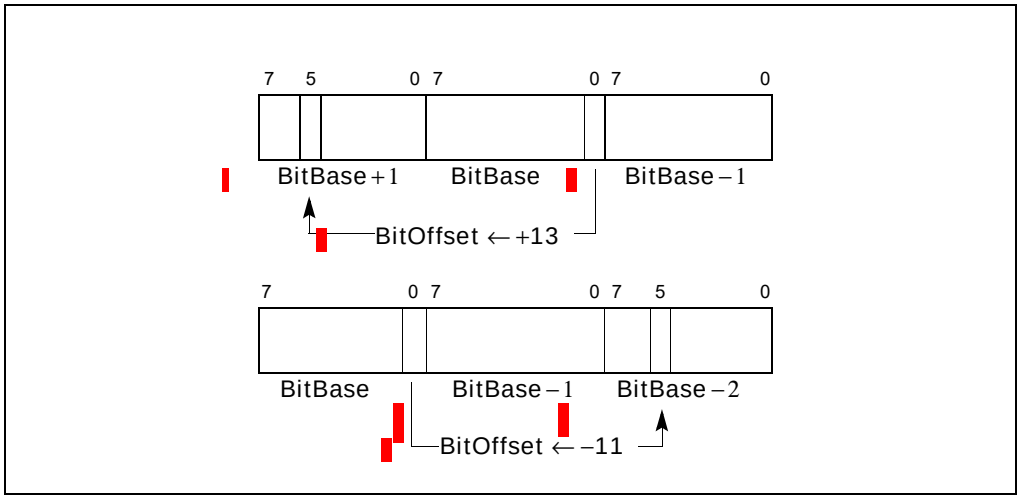


Figure 3-2. Memory Bit Indexing

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.

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	I	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	I	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	I	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	I	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	MI	Valid	Valid	Add with carry *imm8* to *r/m8.*
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	MI	Valid	N.E.	Add with carry *imm8* to *r/m8.*
81 /2 *iw*	ADC *r/m16, imm16*	MI	Valid	Valid	Add with carry *imm16* to *r/m16.*
81 /2 *id*	ADC *r/m32, imm32*	MI	Valid	Valid	Add with CF *imm32* to *r/m32.*
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	MI	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64.*
83 /2 *ib*	ADC *r/m16, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16.*
83 /2 *ib*	ADC *r/m32, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32.*
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	MI	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64.*
10 /*r*	ADC *r/m8, r8*	MR	Valid	Valid	Add with carry byte register to *r/m8.*
REX + 10 /*r*	ADC *r/m8**, *r8**	MR	Valid	N.E.	Add with carry byte register to *r/m64.*
11 /*r*	ADC *r/m16, r16*	MR	Valid	Valid	Add with carry *r16* to *r/m16.*
11 /*r*	ADC *r/m32, r32*	MR	Valid	Valid	Add with CF *r32* to *r/m32.*
REX.W + 11 /*r*	ADC *r/m64, r64*	MR	Valid	N.E.	Add with CF *r64* to *r/m64.*
12 /*r*	ADC *r8, r/m8*	RM	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /*r*	ADC *r8**, *r/m8**	RM	Valid	N.E.	Add with carry *r/m64* to byte register.
13 /*r*	ADC *r16, r/m16*	RM	Valid	Valid	Add with carry *r/m16* to *r16.*
13 /*r*	ADC *r32, r/m32*	RM	Valid	Valid	Add with CF *r/m32* to *r32.*
REX.W + 13 /*r*	ADC *r64, r/m64*	RM	Valid	N.E.	Add with CF *r/m64* to *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA

Description

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

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

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

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

Operation

```
DEST ← DEST + SRC + CF;
```

Intel C/C++ Compiler Intrinsic Equivalent

ADC: extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);

ADC: extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);

ADC: extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);

ADC: extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

ADD—Add

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

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA

Description

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

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

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

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

Operation

DEST ← DEST + SRC;

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

AND—Logical AND

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

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

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

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

Operation

DEST ← DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

BNDLDX—Load Extended Bounds Using Address Translation

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

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

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

Operation

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

Outside 64-bit mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 );
Temp_lb[31:0] ← LoadFrom(A_BTE);
Temp_ub[31:0] ← LoadFrom(A_BTE + 4);
Temp_ptr[31:0] ← LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
```

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

In 64-bit mode

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

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

BNDSTX—Store Extended Bounds Using Address Translation

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

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

Operation

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

Outside 64-bit mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 ); // address of Bound table entry
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
```

In 64-bit mode

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

Intel C/C++ Compiler Intrinsic Equivalent

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

Flags Affected

None

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

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

64-Bit Mode Exceptions

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

BSF—Bit Scan Forward

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BC /r	BSF *r16, r/m16*	RM	Valid	Valid	Bit scan forward on *r/m16.*
0F BC /r	BSF *r32, r/m32*	RM	Valid	Valid	Bit scan forward on *r/m32.*
REX.W + 0F BC /r	BSF *r64, r/m64*	RM	Valid	N.E.	Bit scan forward on *r/m64.*

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

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

Operation

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← 0;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ← temp + 1;
        OD;
        DEST ← temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

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.
#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.
#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.

BSR—Bit Scan Reverse

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BD /r	BSR *r16, r/m16*	RM	Valid	Valid	Bit scan reverse on *r/m16.*
0F BD /r	BSR *r32, r/m32*	RM	Valid	Valid	Bit scan reverse on *r/m32.*
REX.W + 0F BD /r	BSR *r64, r/m64*	RM	Valid	N.E.	Bit scan reverse on *r/m64.*

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

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

Operation

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize – 1;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ← temp - 1;
        OD;
        DEST ← temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

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.
#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.
#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.

CMP—Compare Two Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C *ib*	CMP AL, *imm8*	I	Valid	Valid	Compare *imm8* with AL.
3D *iw*	CMP AX, *imm16*	I	Valid	Valid	Compare *imm16* with AX.
3D *id*	CMP EAX, *imm32*	I	Valid	Valid	Compare *imm32* with EAX.
REX.W + 3D *id*	CMP RAX, *imm32*	I	Valid	N.E.	Compare *imm32 sign-extended to 64-bits* with RAX.
80 /7 *ib*	CMP *r/m8, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m8.*
REX + 80 /7 *ib*	CMP *r/m8**, *imm8*	MI	Valid	N.E.	Compare *imm8* with *r/m8.*
81 /7 *iw*	CMP *r/m16, imm16*	MI	Valid	Valid	Compare *imm16* with *r/m16.*
81 /7 *id*	CMP *r/m32, imm32*	MI	Valid	Valid	Compare *imm32* with *r/m32.*
REX.W + 81 /7 *id*	CMP *r/m64, imm32*	MI	Valid	N.E.	Compare *imm32 sign-extended to 64-bits* with *r/m64.*
83 /7 *ib*	CMP *r/m16, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m16.*
83 /7 *ib*	CMP *r/m32, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m32.*
REX.W + 83 /7 *ib*	CMP *r/m64, imm8*	MI	Valid	N.E.	Compare *imm8* with *r/m64.*
38 /*r*	CMP *r/m8, r8*	MR	Valid	Valid	Compare *r8* with *r/m8.*
REX + 38 /*r*	CMP *r/m8**, *r8**	MR	Valid	N.E.	Compare *r8* with *r/m8.*
39 /*r*	CMP *r/m16, r16*	MR	Valid	Valid	Compare *r16* with *r/m16.*
39 /*r*	CMP *r/m32, r32*	MR	Valid	Valid	Compare *r32* with *r/m32.*
REX.W + 39 /*r*	CMP *r/m64,r64*	MR	Valid	N.E.	Compare *r64* with *r/m64.*
3A /*r*	CMP *r8, r/m8*	RM	Valid	Valid	Compare *r/m8* with *r8.*
REX + 3A /*r*	CMP *r8**, *r/m8**	RM	Valid	N.E.	Compare *r/m8 with r8.*
3B /*r*	CMP *r16, r/m16*	RM	Valid	Valid	Compare *r/m16* with *r16.*
3B /*r*	CMP *r32, r/m32*	RM	Valid	Valid	Compare *r/m32* with *r32.*
REX.W + 3B /*r*	CMP *r64, r/m64*	RM	Valid	N.E.	Compare *r/m64* with *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX (r)	imm8/16/32	NA	NA

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

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 ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

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

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.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#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.

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	ZO	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Instruction Operand Encoding

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

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"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.
02H	EAX	Cache and TLB Information (see Table 3-12).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 217.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		Bits 07 - 05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache. Bits 13 - 10: Reserved. Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***. Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.
	EBX	Bits 11 - 00: L = System Coherency Line Size**. Bits 21 - 12: P = Physical Line partitions**. Bits 31 - 22: W = Ways of associativity**.
	ECX	Bits 31-00: S = Number of Sets**.
	EDX	Bit 00: Write-Back Invalidate/Invalidate. 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31 - 03: Reserved = 0. **NOTES:** * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** Add one to the return value to get the result. ***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache. **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15 - 00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	EBX	Bits 15 - 00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled. Bits 31 - 02: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT. Bits 07 - 04: Number of C1* sub C-states supported using MWAIT. Bits 11 - 08: Number of C2* sub C-states supported using MWAIT. Bits 15 - 12: Number of C3* sub C-states supported using MWAIT. Bits 19 - 16: Number of C4* sub C-states supported using MWAIT. Bits 23 - 20: Number of C5* sub C-states supported using MWAIT. Bits 27 - 24: Number of C6* sub C-states supported using MWAIT. Bits 31 - 28: Number of C7* sub C-states supported using MWAIT. **NOTE:** * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf	
06H	EAX	Bit 00: Digital temperature sensor is supported if set. Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved. Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bits 31 - 15: Reserved.
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31 - 04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0. Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 31 - 04: Reserved = 0.
	EDX	Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)		
07H	Sub-leaf 0 (Input ECX = 0). *	
	EAX	Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. 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.
	ECX	Bit 00: PREFETCHWT1. Bit 01: AVX512_VBMI. Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bits 16 - 5: Reserved. Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID. Supports Read Processor ID if 1. Bits 29 - 23: Reserved. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Reserved. **NOTE:** * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.
	Direct Cache Access Information Leaf	
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.
	EBX	Bit 00: Core cycle event not available if 1. Bit 01: Instruction retired event not available if 1. Bit 02: Reference cycles event not available if 1. Bit 03: Last-level cache reference event not available if 1. Bit 04: Last-level cache misses event not available if 1. Bit 05: Branch instruction retired event not available if 1. Bit 06: Branch mispredict retired event not available if 1. Bits 31 - 07: Reserved = 0.
	ECX	Reserved = 0.
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1). Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14 - 13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31 - 16: Reserved = 0.
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order. For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor. **NOTES:** * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: Invalid. 1: SMT. 2: Core. 3-255: Reserved.
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).
	EAX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04 - 03: MPX state. Bits 07 - 05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 31 - 10: Reserved.
	EBX	Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
	EDX	Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
0DH	EAX	Bit 00: XSAVEOPT is available. Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set. Bit 02: Supports XGETBV with ECX = 1 if set. Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set. Bits 31 - 04: Reserved.
	EBX	Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCR0 | IA32_XSS.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07 - 00: Used for XCR0. Bit 08: PT state. Bit 09: Used for XCR0. Bits 31 - 10: Reserved.
	EDX	Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)		
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 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 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*.
	EBX	Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.
	ECX	Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31 - 02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)		
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved.
L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)		
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31 - 03: Reserved.
Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)		
10H	**NOTES:**	Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31 - 04: Reserved.
	ECX	Reserved.
	EDX	Reserved.
L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)		
10H	**NOTES:**	Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.
	EBX	Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 01- 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)		
10H	**NOTES:**	Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.
	EBX	Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 31 - 00: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)		
10H	**NOTES:**	Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 31 - 00: Reserved.
	ECX	Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
	Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)	
12H		**NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bits 04 - 02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bits 31 - 02: Reserved.
	EBX	Bits 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.
	ECX	Bits 31 - 00: Reserved.
	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.
	Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)	
12H		**NOTES:** Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
	Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)	
12H		**NOTES:** Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.
	EAX	Bit 03 - 00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type	0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Type	0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)	
14H		**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX	Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.
	EBX	Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 31 - 06: Reserved.
	ECX	Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30 - 04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX	Bits 31 - 00: Reserved.
	Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)	
14H	EAX	Bits 02 - 00: Number of configurable Address Ranges for filtering. Bits 15 - 03: Reserved. Bits 31 - 16: Bitmap of supported MTC period encodings.
	EBX	Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings. Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.
	ECX	Bits 31 - 00: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 31 - 00: Reserved.
	Time Stamp Counter and Nominal Core Crystal Clock Information Leaf	
15H		**NOTES:** If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. If ECX is 0, the nominal core crystal clock frequency is not enumerated. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.
	EAX	Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31 - 00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31 - 00: Reserved = 0.
	Processor Frequency Information Leaf	
16H	EAX	Bits 15 - 00: Processor Base Frequency (in MHz). Bits 31 - 16: Reserved =0.
	EBX	Bits 15 - 00: Maximum Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	ECX	Bits 15 - 00: Bus (Reference) Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EDX	Reserved.
		NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.
	System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)	
17H		**NOTES:** Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.
	EAX	Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX	Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.
	ECX	Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX	Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)		
17H	EAX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
		NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.
System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)		
17H		**NOTES:** Leaf 17H output depends on the initial value in ECX.
	EAX	Bits 31 - 00: Reserved = 0.
	EBX	Bits 31 - 00: Reserved = 0.
	ECX	Bits 31 - 00: Reserved = 0.
	EDX	Bits 31 - 00: Reserved = 0.
Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)		
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 *Intel® 64 and IA-32 Architectures Optimization Reference Manual* 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.
	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.
	ECX	Bits 31 - 00: S = Number of Sets.

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. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: 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.
	Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)	
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 *Intel® 64 and IA-32 Architectures Optimization Reference Manual* for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reserved.
	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.
	ECX	Bits 31 - 00: S = Number of Sets.
	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.
	Unimplemented CPUID Leaf Functions	
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode. Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET available in 64-bit mode. Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.
80000006H	EAX EBX	Reserved = 0. Reserved = 0.
	ECX	Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units.
	EDX	Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		NOTES: * L2 associativity field encodings: 00H - Disabled. 01H - Direct mapped. 02H - 2-way. 04H - 4-way. 06H - 8-way. 08H - 16-way. 0FH - Fully associative.
80000007H	EAX 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 EBX ECX EDX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0. **NOTES:** * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (* "Genu", with G in the low eight bits of BL *)
EDX ← 49656e69h (* "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.

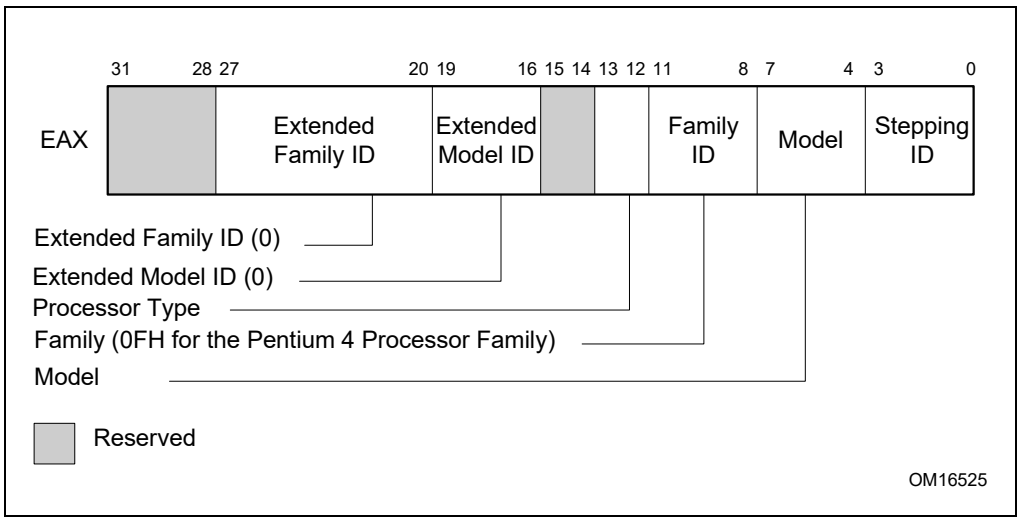


Figure 3-6. Version Information Returned by CPUID in EAX

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

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID « 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

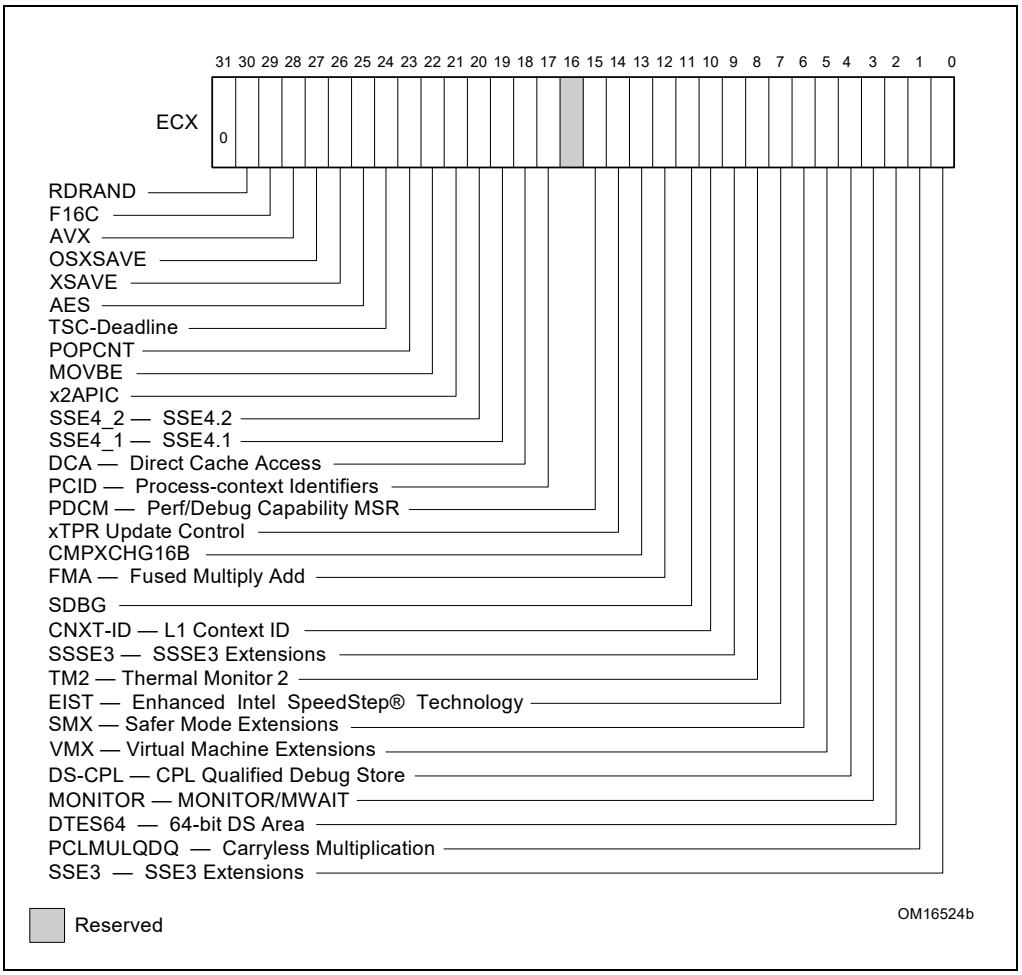


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	**PCLMULQDQ.** A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions.** A value of 1 indicates that the processor supports this technology.
6	SMX	**Safer Mode Extensions.** A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.
7	EIST	**Enhanced Intel SpeedStep® technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

Table 3-10. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	**Perfmon and Debug Capability:** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
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.

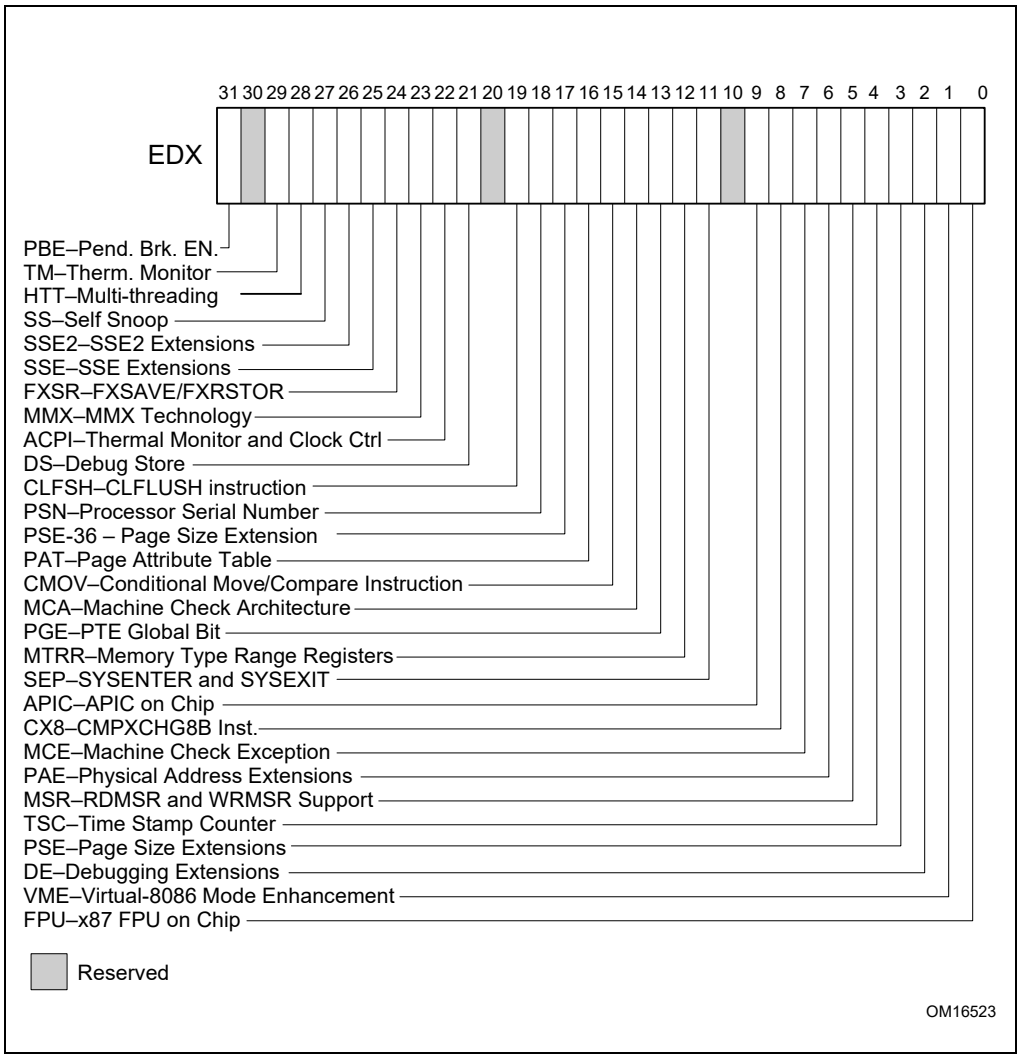


Figure 3-8. Feature Information Returned in the EDX Register

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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Max APIC IDs reserved field is Valid.** A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-12. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K-µop, 8-way set associative
71H	Cache	Trace cache: 16 K-µop, 8-way set associative
72H	Cache	Trace cache: 32 K-µop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GBbyte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
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-µop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

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

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

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

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

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, “Introduction to Virtual-Machine Extensions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
    IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
        Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
    FI;
```

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit

1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

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.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The Processor Brand String Method

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.

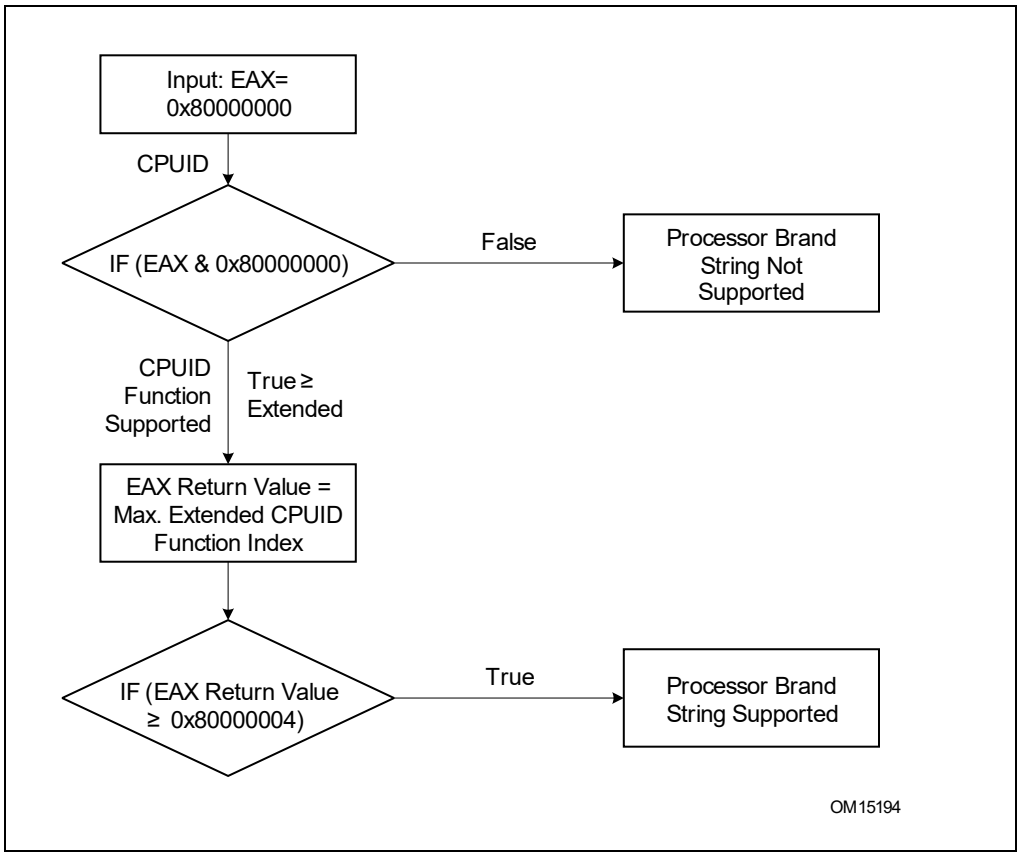


Figure 3-9. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-13. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	“ ” “ ” “ ” “nI ”
80000003H	EAX = 286C6574H 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.

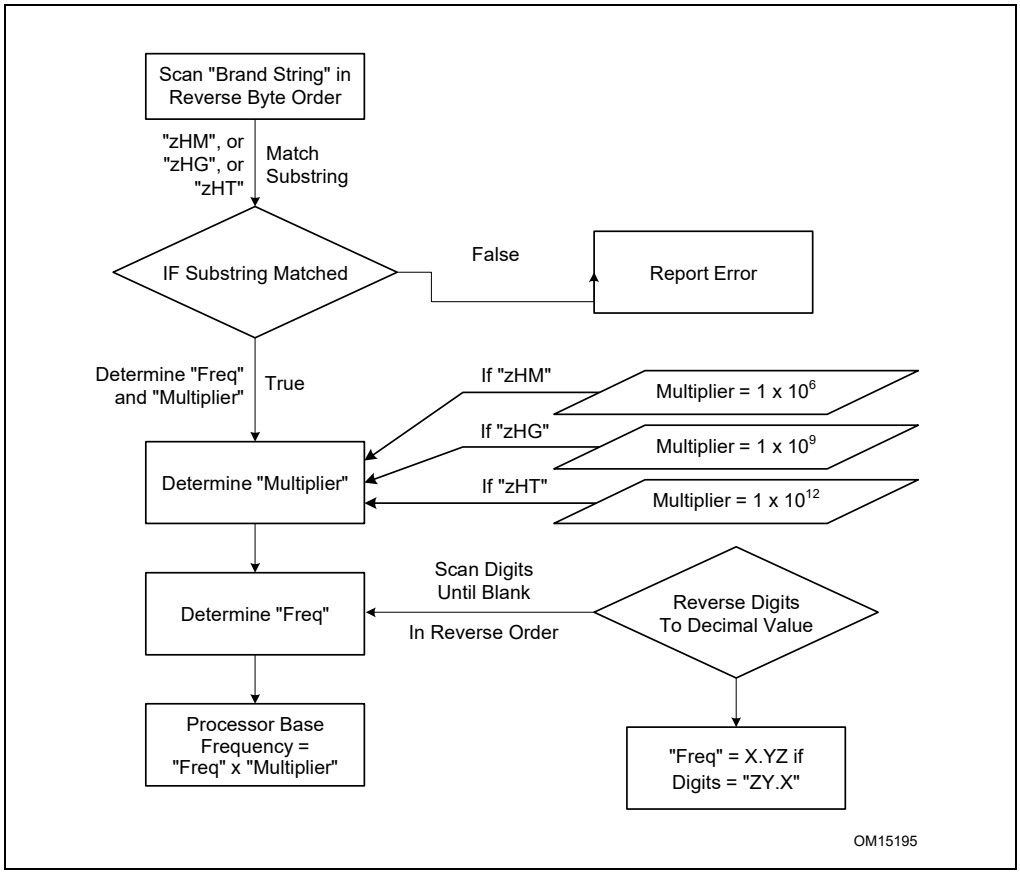


Figure 3-10. Algorithm for Extracting Processor Frequency

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 = 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.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /4	FBLD *m80bcd*	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

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

Operation

```
TOP ← TOP – 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

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.
#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.
#MF	If there is a pending x87 FPU exception.
#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.

11. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, M-U.

Changes to this chapter:

Updates to the following instructions are covered here with change bars: MOV, MOVDQU/VMOVDQU8/16/32/64, MOVNTDQA, MOVQ, MOVQ2DQ, PADDB/PADDW/PADDD/PADDQ, PCMPGTB/PCMPGTW/PCMPGTD, PMADDUBSW, PMADDWD, PSLLW/PSLLD/PSLLQ, PSRLW/PSRLD/PSRLQ, RDTSC, RDTSCP, SFENCE, SUB, SYSRET, and UD.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8.*
REX + 88 /r	MOV *r/m8****,*r8****	MR	Valid	N.E.	Move *r8* to *r/m8.*
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16.*
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32.*
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64.*
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8.*
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8.*
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16.*
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32.*
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64.*
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16.*
REX.W + 8C /r	MOV *r16/r32/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r16/r32/r64/m16.*
REX.W + 8C /r	MOV *r64/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r64/m16.*
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8.*
REX + B0+ *rb ib*	MOV *r8****, *imm8*	OI	Valid	N.E.	Move *imm8* to *r8.*
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16.*
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32.*
REX.W + B8+ *rd io*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64.*
C6 */0 ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8.*
REX + C6 */0 ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8.*
C7 */0 iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16.*
C7 */0 id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32.*
REX.W + C7 */0 id*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64.*

NOTES:

* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs[1]. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium

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

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

   MOV SS, EDX
   MOV SS, EAX
   MOV ESP, EBP

Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium and earlier processors.

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;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        OR segment selector's RPL ≠ CPL
        OR segment is not a writable data segment
        OR DPL ≠ CPL
            THEN #GP(selector); FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS ← segment selector;
                SS ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
    IF segment selector index is outside descriptor table limits
    OR segment is not a data or readable code segment
    OR ((segment is a data or nonconforming code segment) AND ((RPL > DPL) or (CPL > DPL)))
        THEN #GP(selector); FI;
    IF segment not marked present
        THEN #NP(selector);
        ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with NULL selector
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, and either the RPL or the CPL is greater than the DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked 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.
#UD	If attempt is made to load the CS register.
	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 attempt is made to load the CS register.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If an attempt is made to load SS register with NULL segment selector when CPL = 3.
	If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked 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.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned 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 byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) 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.

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.

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

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

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

Operation

VMOVDQU8 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0           ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU8 (EVEX encoded versions, store-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i]←
            SRC[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX encoded versions, load-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, store-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i]←
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX encoded versions, load-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU32 (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
```

VMOVDQU32 (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;
```

VMOVDQU32 (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
```

VMOVDQU64 (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
```

VMOVDQU64 (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;
```

VMOVDQU64 (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
```

VMOVDQU (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVDQU (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

```
VMOVDQU (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

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

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

(V)MOVDQU (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16(__mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16(__mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16(__mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32(void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32(__mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQU32 __m128i _mm_maskz_loadu_epi32(__mmask8 k, void * sa);

VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64(void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	AVX512VL AVX512F	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	AVX512VL AVX512F	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

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

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

• A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.

• A non-WC reference to memory already resident in a temporary internal buffer.

• Interleaving of reads and writes to a single temporary internal buffer.

• Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.

• Certain micro-architectural conditions including resource shortages, detection of

a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read 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 read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

1. ModRM.MOD = 011B required

tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE form)
DEST ←SRC
DEST[MAXVL-1:128] (Unmodified)

VMOVNTDQA (VEX.128 and EVEX.128 encoded form)
DEST ← SRC
DEST[MAXVL-1:128] ← 0

VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVNTDQA (EVEX.512 encoded form)
DEST[511:0] ← SRC[511:0]
DEST[MAXVL-1:512] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ *mm, mm/m64*	A	V/V	MMX	Move quadword from *mm/m64* to *mm*.
NP 0F 7F /r MOVQ *mm/m64, mm*	B	V/V	MMX	Move quadword from *mm* to *mm/m64*.
F3 0F 7E /r MOVQ *xmm1, xmm2/m64*	A	V/V	SSE2	Move quadword from *xmm2/mem64* to *xmm1*.
VEX.128.F3.0F.WIG 7E /r VMOVQ *xmm1, xmm2/m64*	A	V/V	AVX	Move quadword from *xmm2* to *xmm1*.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ *xmm2/m64, xmm1*	B	V/V	SSE2	Move quadword from *xmm1* to *xmm2/mem64*.
VEX.128.66.0F.WIG D6 /r VMOVQ *xmm1/m64, xmm2*	B	V/V	AVX	Move quadword from *xmm2* register to *xmm1/m64*.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F	Move quadword from xmm2 register to xmm1/m64.

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	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

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

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ instruction when operating on MMX technology registers and memory locations

DEST ← SRC;

MOVQ instruction when source and destination operands are XMM registers

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;

MOVQ instruction when source operand is XMM register and destination
operand is memory location:

DEST ← SRC[63:0];

MOVQ instruction when source operand is memory location and destination
operand is XMM register:

DEST[63:0] ← SRC;
DEST[127:64] ← 0000000000000000H;

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with XMM register source and destination
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E) with memory source
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with memory source
DEST[63:0] ← SRC[63:0]
DEST[:MAXVL-1:64] ← 0

VMOVQ (D6) with memory dest
DEST[63:0] ← SRC2[63:0]

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ __m128i _mm_loadu_si64(void * s);
VMOVQ void _mm_storeu_si64(void * d, __m128i s);

MOVQ m128i _mm_move_epi64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F D6 /r	MOVQ2DQ *xmm, mm*	RM	Valid	Valid	Move quadword from *mmx* to low quadword of *xmm*.

Instruction Operand Encoding

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

Description

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.

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FC /r[1] PADDB *mm, mm/m64*	A	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
NP 0F FD /r[1] PADDW *mm, mm/m64*	A	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
NP 0F FE /r[1] PADDD *mm, mm/m64*	A	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
NP 0F D4 /r[1] PADDQ *mm, mm/m64*	A	V/V	MMX	Add packed quadword integers from *mm/m64* and *mm*.
66 0F FC /r PADDB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W0 FE /r VPADDD *xmm1 {k1}{z}, xmm2,* xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *xmm2, and xmm3/m128/m32bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W1 D4 /r VPADDQ *xmm1 {k1}{z}, xmm2,* xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *xmm2, and xmm3/m128/m64bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F.W0 FE /r VPADDD *ymm1 {k1}{z}, ymm2,* ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *ymm2, ymm3/m256/m32bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ *ymm1 {k1}{z}, ymm2,* ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *ymm2, ymm3/m256/m64bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FC /r VPADDB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed byte integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FD /r VPADDW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed word integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	D	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ *zmm1 {k1}{z}, zmm2,* zmm3/m512/m64bcst	D	V/V	AVX512F	Add packed quadword integers from *zmm2, zmm3/m512/m64bcst* and store in *zmm1* using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too

large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

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

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (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 register destination are unmodified.

Operation

PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

PADDW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

PADDD (with 64-bit operands)

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

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE instruction)

```
DEST[7:0]← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← DEST[127:120] + SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← DEST[127:112] + SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE instruction)

```
DEST[31:0]← DEST[31:0]  + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96]← DEST[127:96] + SRC[127:96];
DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE instruction)

```
DEST[63:0]← DEST[63:0]  + SRC[63:0];
DEST[127:64]← DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] ← 0;
```

VPADDW (VEX.128 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← SRC1[127:112] + SRC2[127:112];
DEST[MAXVL-1:128] ← 0;
```

VPADDD (VEX.128 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← SRC1[127:96] + SRC2[127:96];
DEST[MAXVL-1:128] ← 0;
```

VPADDQ (VEX.128 encoded instruction)

```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] ← 0;
```

VPADDB (VEX.256 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248]← SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240]← SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] ← SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 encoded instruction)

```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[191:128]← SRC1[191:128]  + SRC2[191:128];
DEST[255:192] ← SRC1[255:192] + SRC2[255:192];
```

VPADDB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)

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

VPADDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPADDB__m512i _mm512_add_epi8 (__m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 (__m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32(__m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32(__mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32(__mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64(__m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64(__mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64(__mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64(__mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b);
PADDW __m128i _mm_add_epi16 (__m128i a, __m128i b);
PADDD __m128i _mm_add_epi32 (__m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 (__m128i a, __m128i b);

VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b);
VPADDW __m256i _mm256_add_epi16 (__m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 (__m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 (__m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDD/Q, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB *mm, mm/m64*	A	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /r PCMPGTB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 65 /r[1] PCMPGTW *mm, mm/m64*	A	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /r PCMPGTW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 66 /r[1] PCMPGTD *mm, mm/m64*	A	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /r PCMPGTD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed word integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed doubleword integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed word integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed doubleword integers in *ymm2* and *ymm3/m256* for greater than.
EVEX.NDS.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

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

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (with 64-bit operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] ← FFH;
        ELSE DEST[63:56] ← 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPGTB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTB (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTW (with 64-bit operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW (with 128-bit operands)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTW (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
```

```
        ELSE      DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTD (with 64-bit operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; FI;
```

PCMPGTD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE      DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB:__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW:__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD:__m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB:__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW:__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD:__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB: __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW: __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD: __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTD, see Exceptions Type E4.

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r[1] PMADDUBSW *mm1, mm2/m64*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *mm1*.
66 0F 38 04 /r PMADDUBSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *ymm1*.
EVEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (with 64 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (with 128 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] ← 0
```

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] ← 0
```

VPMADDUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_maddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW: __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW: __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD *mm, mm/m64*	A	V/V	MMX	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r PMADDWD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed word integers in *xmm2* by the packed word integers in *xmm3/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed word integers in *ymm2* by the packed word integers in *ymm3/m256*, add adjacent doubleword results, and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent double-word results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the double-word results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

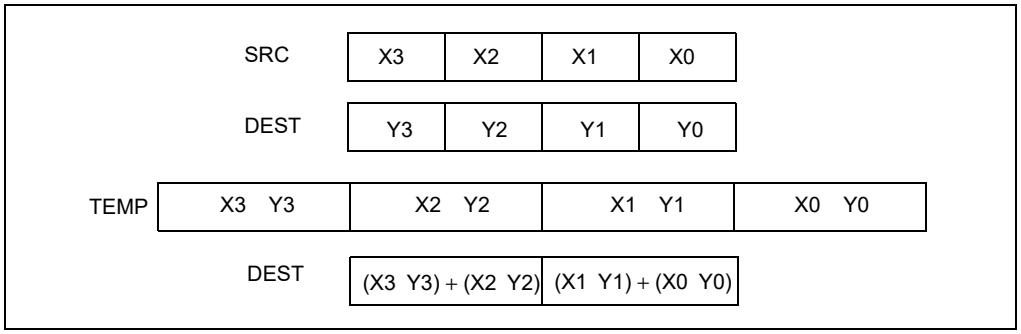


Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

```
DEST[31:0]    (DEST[15:0]   SRC[15:0]) + (DEST[31:16]   SRC[31:16]);
DEST[63:32]    (DEST[47:32]   SRC[47:32]) + (DEST[63:48]   SRC[63:48]);
```

PMADDWD (with 128-bit operands)

```
DEST[31:0]    (DEST[15:0]   SRC[15:0]) + (DEST[31:16]   SRC[31:16]);
DEST[63:32]    (DEST[47:32]   SRC[47:32]) + (DEST[63:48]   SRC[63:48]);
DEST[95:64]    (DEST[79:64]   SRC[79:64]) + (DEST[95:80]   SRC[95:80]);
DEST[127:96]    (DEST[111:96]   SRC[111:96]) + (DEST[127:112]   SRC[127:112]);
```

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] ← 0
```

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r PSLLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 71 /6 ib PSLLW *mm1, imm8*	B	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F2 /r[1] PSLLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r PSLLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1, imm8*	B	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F3 /r[1] PSLLQ *mm, mm/m64*	A	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r PSLLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ *mm, imm8*	B	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW *xmm1, xmm2, xmm3/m128* VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW *xmm1, xmm2, imm8*	C	V/V	AVX	Shift words in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* left by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.NDS.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

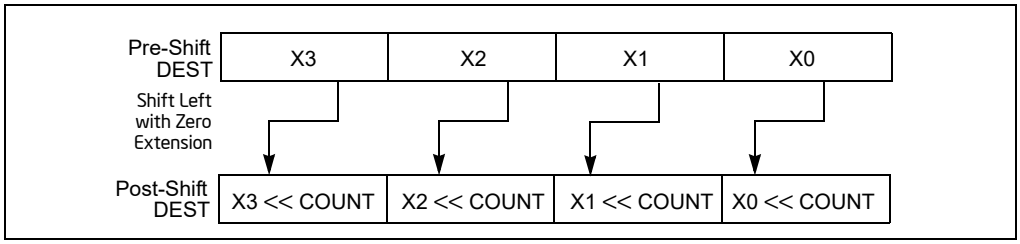


Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
```

PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST ← ZeroExtend(DEST << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
```

```
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
```

```
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←000000000000000000000000000000000H
    DEST[255:128] ←000000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←000000000000000000000000000000000H
    DEST[255:128] ←000000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

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

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;

VPSLLD (ymm, imm8) - VEX.256 encoding
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0

VPSLLD (xmm, imm8) - VEX.128 encoding
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSLLQ (EVEX versions, imm8)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VPSLLQ (EVEX versions, xmm/m128)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);

VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW:__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW:__m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW:__m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW:__m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW:__m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW:__m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD:__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD:__m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD:__m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD:__m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD:__m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD:__m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ:__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ:__m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ:__m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ:__m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ:__m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ:__m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r PSRLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D2 /r[1] PSRLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r PSRLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D3 /r[1] PSRLQ *mm, mm/m64*	A	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /r PSRLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ *mm, imm8*	B	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift quadwords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ *xmm1, xmm2, imm8*	D	V/V	AVX	Shift quadwords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* right by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	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:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

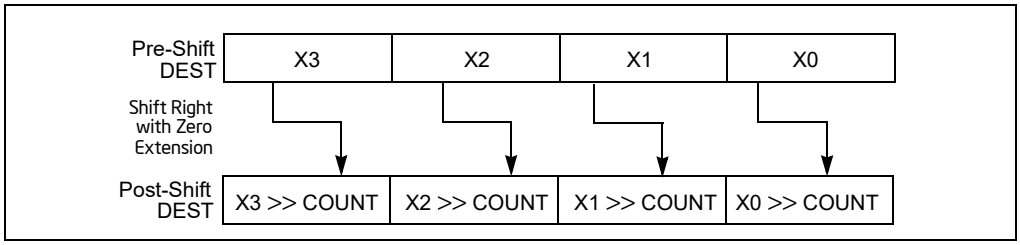


Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (with 64-bit operand)

```
    IF (COUNT > 15)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
    FI;
```

PSRLD (with 64-bit operand)

```
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (with 64-bit operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST ← ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
```

```
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ←0
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ←0
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ←0
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

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

VPSRLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLD (xmm, imm8) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX versions, xmm/m128)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC1 *is memory*)
                    THEN DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                    ELSE DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
                FI;
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i _mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);

VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW:__m64 _mm_srli_pi16(__m64 m, int count)
PSRLW:__m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW:__m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW:__m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW:__m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW:__m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD:__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD:__m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD:__m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD:__m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD:__m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD:__m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ:__m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ:__m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ:__m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ:__m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ:__m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ:__m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:
Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.
Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:
Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.
Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

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

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSC:

- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads are globally visible,[1] it can execute LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads and stores are globally visible, it can execute the sequence MFENCE;LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute the sequence LFENCE immediately after RDTSC.

This instruction was introduced by the Pentium processor.

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 (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#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 TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F9	RDTSCP	ZO	Valid	Valid	Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

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

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible.[1] But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSCP:

- If software requires RDTSCP to be executed only after all previous stores are globally visible, it can execute MFENCE immediately before RDTSCP.
- If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP.

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 (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

SFENCE—Store Fence

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F8	SFENCE	ZO	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

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

Description

Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_sfence(void)

Exceptions (All Operating Modes)

#UD	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C *ib*	SUB AL, i*mm8*	I	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, i*mm16*	I	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, i*mm32*	I	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, i*mm32*	I	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from RAX.
80 /5 *ib*	SUB *r/m8, imm8*	MI	Valid	Valid	Subtract *imm8* from *r/m8.*
REX + 80 /5 *ib*	SUB *r/m8*, imm8*	MI	Valid	N.E.	Subtract *imm8* from *r/m8.*
81 /5 *iw*	SUB *r/m16, imm16*	MI	Valid	Valid	Subtract *imm16* from *r/m16.*
81 /5 *id*	SUB *r/m32, imm32*	MI	Valid	Valid	Subtract *imm32* from *r/m32.*
REX.W + 81 /5 *id*	SUB *r/m64, imm32*	MI	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from *r/m64.*
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*
83 /5 *ib*	SUB *r/m32, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m32.*
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	MI	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64.*
28 /*r*	SUB *r/m8, r8*	MR	Valid	Valid	Subtract *r8* from *r/m8.*
REX + 28 /*r*	SUB *r/m8*, r8**	MR	Valid	N.E.	Subtract *r8* from *r/m8.*
29 /*r*	SUB *r/m16, r16*	MR	Valid	Valid	Subtract *r16* from *r/m16.*
29 /*r*	SUB *r/m32, r32*	MR	Valid	Valid	Subtract *r32* from *r/m32.*
REX.W + 29 /*r*	SUB *r/m64, r64*	MR	Valid	N.E.	Subtract *r64* from *r/m64.*
2A /*r*	SUB *r8, r/m8*	RM	Valid	Valid	Subtract *r/m8* from *r8.*
REX + 2A /*r*	SUB *r8*, r/m8**	RM	Valid	N.E.	Subtract *r/m8* from *r8.*
2B /*r*	SUB *r16, r/m16*	RM	Valid	Valid	Subtract *r/m16* from *r16.*
2B /*r*	SUB *r32, r/m32*	RM	Valid	Valid	Subtract *r/m32* from *r32.*
REX.W + 2B /*r*	SUB *r64, r/m64*	RM	Valid	N.E.	Subtract *r/m64* from *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

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

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

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

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

Operation

DEST ← (DEST – SRC);

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	ZO	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	ZO	Valid	Invalid	Return to 64-bit mode from fast system call

Instruction Operand Encoding

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

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.[1] With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 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 SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch 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,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).
- 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.

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

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;
IF (CPL ≠ 0) THEN #GP(0); FI;

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        IF (RCX is not canonical) THEN #GP(0);
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
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 ← FFFFFH;                     (* 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 Compatibility Mode *)
        CS.L ← 0;                      (* Compatibility mode *)
        CS.D ← 1;                      (* 32-bit code segment *)
FI;
CS.G ← 1;                              (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3;  (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                           (* Flat segment *)
SS.Limit ← FFFFFH;                     (* 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

All.

Protected Mode Exceptions

#UD	The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
#GP(0)	If CPL ≠ 0.
	If the return is to 64-bit mode and RCX contains a non-canonical address.

UD—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F FF /r	UD0[1] *r32, r/m32*	RM	Valid	Valid	Raise invalid opcode exception.
0F B9 /r	UD1 *r32, r/m32*	RM	Valid	Valid	Raise invalid opcode exception.
0F 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.

NOTES:

1. Some older processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

Instruction Operand Encoding

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

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

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

Operation

```
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	Raises an invalid opcode exception in all operating modes.

12.Updates to Chapter 5, Volume 2C

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C:* Instruction Set Reference, V-Z.

Changes to this chapter:

Updates to the following instructions are covered here with change bars: VBROADCAST, VCVTQQ2PD, VGATHERDPS/VGATHERQPS, VMASKMOV, VPBROADCAST, VPCONFLICTD/Q, VPGATHERDD/VPGATHERQD, VPMOVWB/VPMOVSWB/VPMOVUSWB, VPMULTISHIFTQB, VSCALEFPD, VSCALEFPS, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XTEST.

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VEX.128.66.0F38.W0 18/r VBROADCASTSS *xmm1, xmm2*	A	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS *ymm1, xmm2*	A	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD *ymm1, xmm2*	A	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 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	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

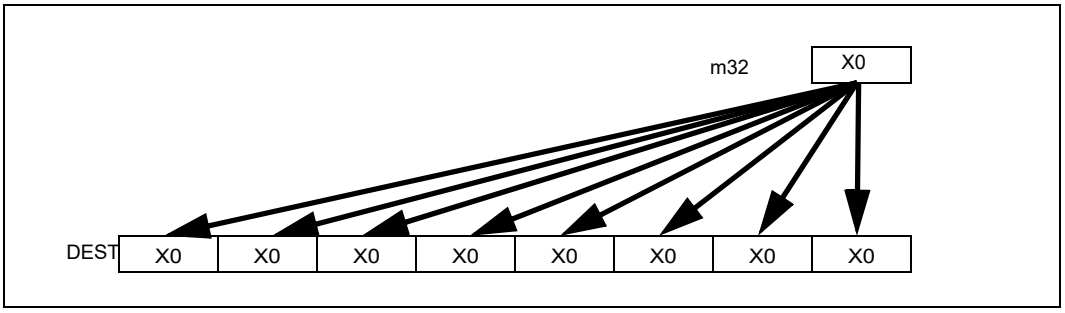


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

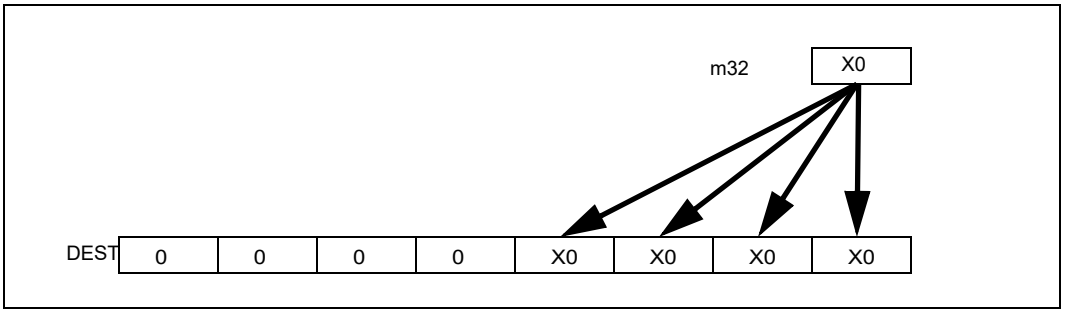


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

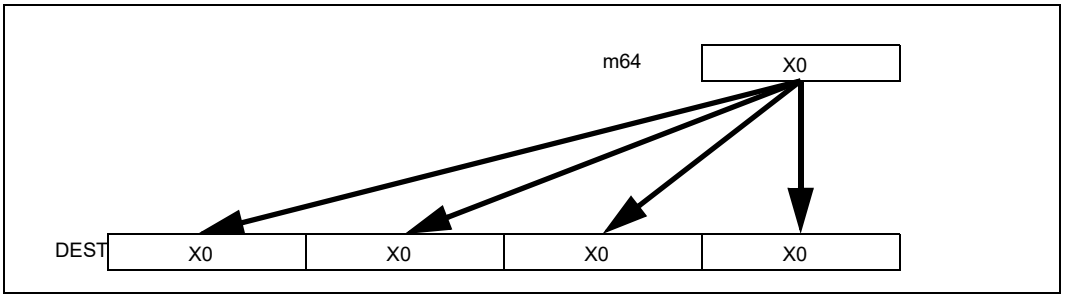


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

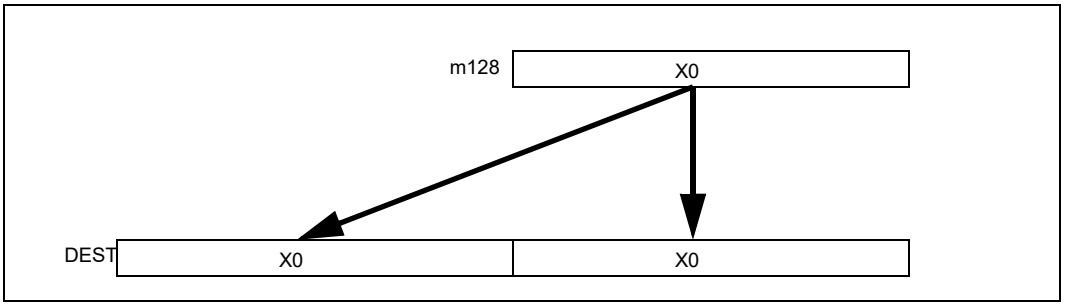


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

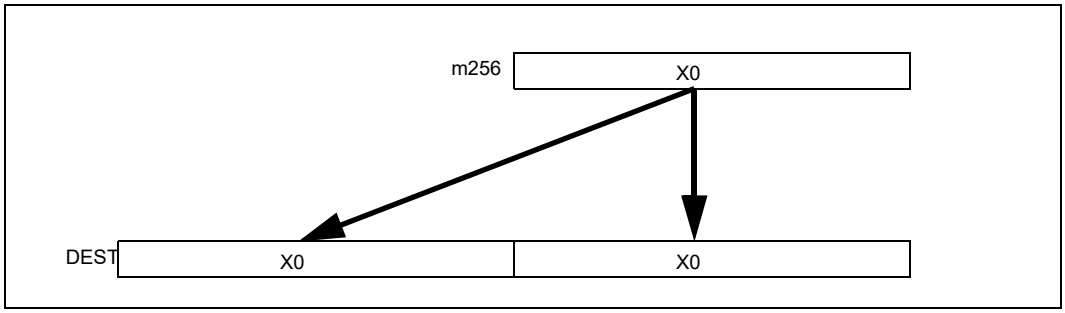


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

Operation

VBROADCASTSS (128 bit version VEX and legacy)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAXVL-1:128] ← 0
```

VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTSS (EVEX encoded versions)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTSD (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTSD (EVEX encoded versions)

```
(KL, VL) = (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[63:0]
        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
```

VBROADCASTF32x2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j * 32
    n ←   (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTF32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j* 32
    n ←   (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF64X2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    n ←(j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTF32X8 (EVEX.U1.512 encoded version)

```
FOR j ← 0 TO 15
    i ← j * 32
    n ←(j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF64X4 (EVEX.512 encoded version)

```
FOR j ← 0 TO 7
    i ← j * 64
    n ←    (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8(__m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8(__mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4(__m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4(__mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd(__m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps(__m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcastss_ps(__mmask8 k, __m128 a);

VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6.

#UD	If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
	If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
	If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/m512/m64bcst 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	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        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
```

VCVTQQ2PD (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
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD __m512d _mm512_cvtepi64_pd(__m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd(__m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd(__mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd(__m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvt_roundepi64_pd(__m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvt_roundepi64_pd(__mmask8 k, __m512i a, int r);
VCVTQQ2PD __m256d _mm256_mask_cvtepi64_pd(__m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d _mm256_maskz_cvtepi64_pd(__mmask8 k, __m256i a);
VCVTQQ2PD __m128d _mm_mask_cvtepi64_pd(__m128d s, __mmask8 k, __m128i a);
VCVTQQ2PD __m128d _mm_maskz_cvtepi64_pd(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD	If EVEX.vvvv != 1111B.

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS *xmm1, vm32x, xmm2*	A	V/V	AVX2	Using dword indices specified in *vm32x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64x, xmm2*	A	V/V	AVX2	Using qword indices specified in *vm64x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS *ymm1, vm32y, ymm2*	A	V/V	AVX2	Using dword indices specified in *vm32y*, gather single-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64y, xmm2*	A	V/V	AVX2	Using qword indices specified in *vm64y*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VGATHERDPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERDPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);

VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);

VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);

VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERQPS: __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);

VGATHERQPS: __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *ymm2* using mask in *ymm1*.
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *ymm2* using mask in *ymm1*.

Instruction Operand Encoding

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

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0
```

VMASKMOVPS - 256-bit load

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

VMASKMOVPD - 128-bit load

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

VMASKMOVPD - 256-bit load

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

VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

VMASKMOVPS - 256-bit store

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

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

__m256d _mm256_maskload_pd(double *a, __m256i mask);

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

__m128 _mm_maskload_ps(float const *a, __m128i mask)

void _mm_maskstore_ps(float *a, __m128i mask, __m128 b)

__m128d _mm_maskload_pd(double *a, __m128i mask);

void _mm_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations); additionally

#UD	If VEX.W = 1.

VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	B	V/V	AVX512BW	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	B	V/V	AVX512BW	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to two locations in xmm1.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	A	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 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	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

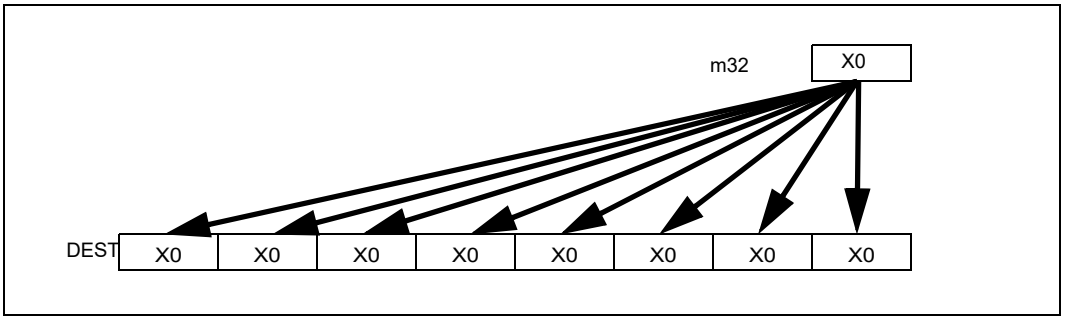


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

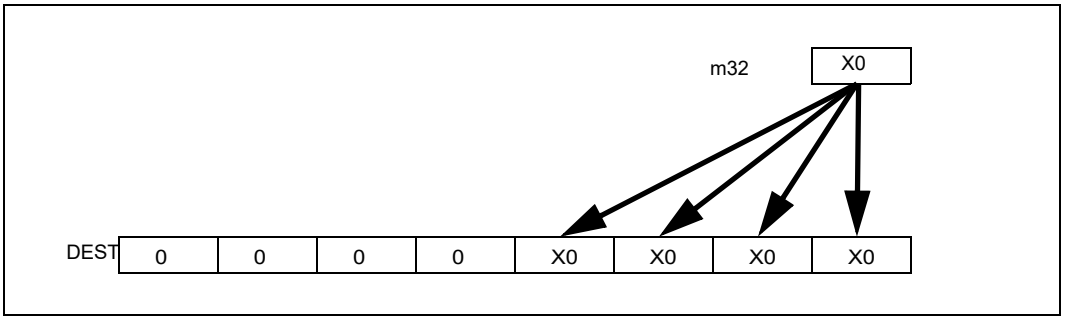


Figure 5-17. VPBROADCASTD Operation (128-bit version)

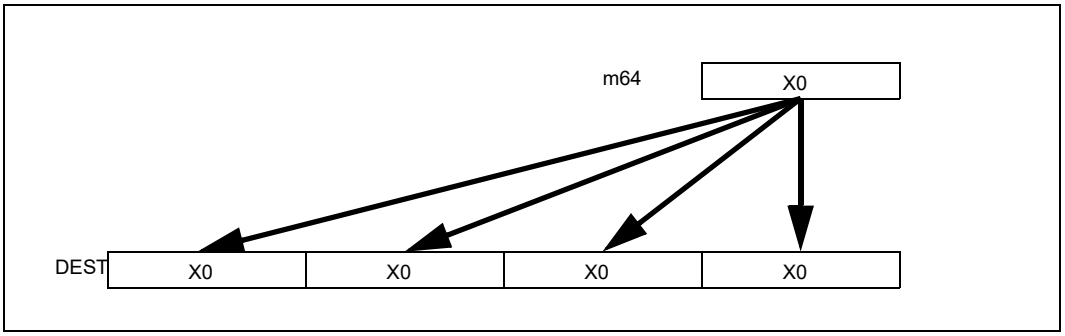


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

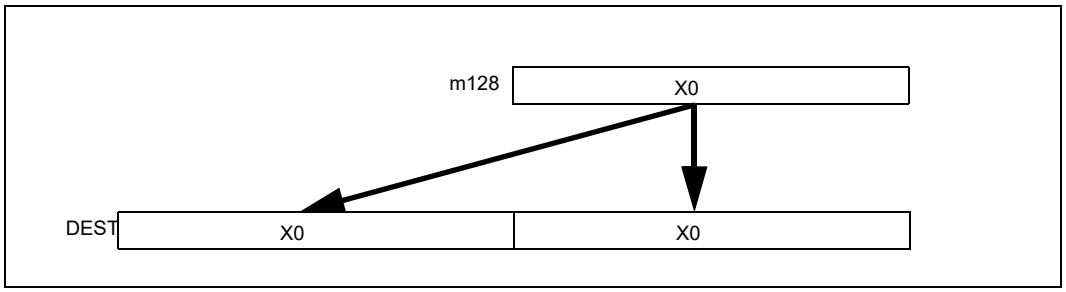


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

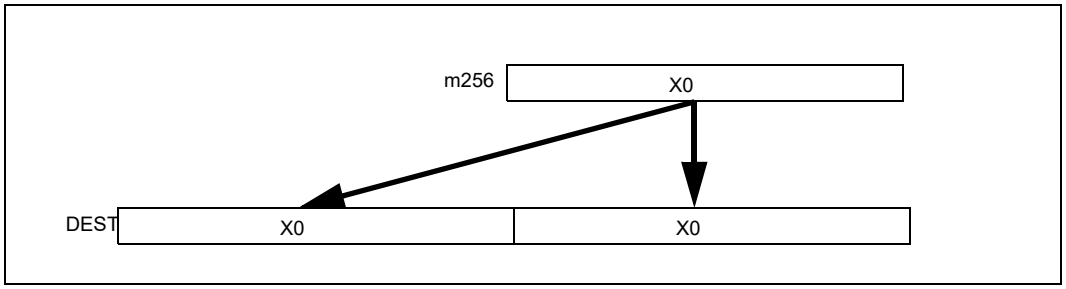


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ←j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[7:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ←j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTD (128 bit version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAXVL-1:128] ← 0
```

VPBROADCASTD (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAXVL-1:256] ← 0
```

```
VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTQ (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0
```

VPBROADCASTQ (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 DEST[i+63:i] ← SRC[63:0]
        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
VBROADCASTI32x2 (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j * 32
    n ←   (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTI32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j* 32
    n ←   (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI64X2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    n ←(j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTI32X8 (EVEX.U1.512 encoded version)

```
FOR j ← 0 TO 15
    i ← j * 32
    n ←(j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI64X4 (EVEX.512 encoded version)

```
FOR j ← 0 TO 7
    i ← j * 64
    n ←   (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_broadcastb_epi8(__m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8(__mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i _mm512_broadcastd_epi32(__m128i a);
VPBROADCASTD __m512i _mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i _mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
VPBROADCASTD __m256i _mm256_broadcastd_epi32(__m128i a);
VPBROADCASTD __m256i _mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i _mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i _mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_broadcast_i32x2(__m128i a);

VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m256i _mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i _mm_mask_broadcast_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i _mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m256i _mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i _mm512_broadcast_i32x8(__m256i a);
VBROADCASTI32x8 __m512i _mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBROADCASTI32x8 __m512i _mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i _mm512_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m512i _mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m512i _mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m256i _mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x4 __m512i _mm512_broadcast_i64x4(__m256i a);
VBROADCASTI64x4 __m512i _mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBROADCASTI64x4 __m512i _mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
	If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.
	If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

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

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPCONFLICTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*THEN
        FOR k ← 0 TO j-1
            m ← k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] ← 1
            ELSE
                DEST[i+k] ← 0
            FI
        ENDFOR
        DEST[i+31:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPCONFLICTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
     i ← j*64
     IF MaskBit(j) OR *no writemask*THEN
        FOR k ← 0 TO j-1
            m ← k*64
             IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] ← 1
             ELSE
                DEST[i+k] ← 0
            FI
         ENDFOR
         DEST[i+63:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
             *DEST[i+63:i] remains unchanged*
         ELSE
             DEST[i+63:i] ← 0
         FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4NF.

VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD *ymm1, vm32y, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32y*, gather dword from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64y, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDD (VEX.128 version)
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
```

VPGATHERQD (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);

VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed word integers from *xmm2* into 8 packed bytes in *xmm1/m64* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed signed word integers from *xmm2* into 8 packed signed bytes in *xmm1/m64* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed unsigned word integers from *xmm2* into 8 packed unsigned bytes in *8mm1/m64* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed word integers from *ymm2* into 16 packed bytes in *xmm1/m128* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed signed word integers from *ymm2* into 16 packed signed bytes in *xmm1/m128* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed unsigned word integers from *ymm2* into 16 packed unsigned bytes in *xmm1/m128* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed word integers from *zmm2* into 32 packed bytes in *ymm1/m256* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed signed word integers from *zmm2* into 32 packed signed bytes in *ymm1/m256* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed unsigned word integers from *zmm2* into 32 packed unsigned bytes in *ymm1/m256* using unsigned saturation under writemask k1.

Instruction Operand Encoding

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

Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
        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/2] ← 0;
```

VPMOVWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
        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/2] ← 0;
```

VPMOVSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
        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/2] ← 0;
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVUSWB __m256i _mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i _mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtusepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i _mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVSWB void _mm512_mask_cvtsepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void _mm512_mask_cvtepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVUSWB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtusepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtusepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i _mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVSWB void _mm256_mask_cvtsepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i _mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void _mm_mask_cvtsepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i _mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void _mm256_mask_cvtepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void _mm_mask_cvtepi16_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_VBMI AVX512VL	Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.
EVEX.NDS.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	AVX512_VBMI AVX512VL	Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.
EVEX.NDS.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_VBMI	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

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.

Operation

VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)

```
(KL, VL) = (2, 128),(4, 256), (8, 512)
FOR i ← 0 TO KL-1
    IF EVEX.b=1 AND src2 is memory THEN
            tcur ← src2.qword[0]; //broadcasting
    ELSE
            tcur ← src2.qword[i];
    FI;
    FOR j ← 0 to 7
        ctrl ← src1.qword[i].byte[j] & 63;
        FOR k ← 0 to 7
            res.bit[k] ← tcur.bit[ (ctrl+k) mod 64 ];
        ENDFOR
        IF k1[i*8+j] or no writemask THEN
            DEST.qword[i].byte[j] ← res;
        ELSE IF zeroing-masking THEN
            DEST.qword[i].byte[j] ← 0;
    ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8(__m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8(__mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8(__m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8(__mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8(__m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	A	V/V	AVX512F	Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-21 and Table 5-22.

Table 5-21. \VSCALEFPD/SD/PS/SS Special Cases

		Src2				Set IE
		±NaN	+Inf	-Inf	0/Denorm/Norm	
Src1	±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	±Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 5-22. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{1024}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
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] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
                ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	A	V/V	AVX512F	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-21 and Table 5-23.

Table 5-23. Additional VSCALEFPS/SS Special Cases

Special Case	Returned value	Faults
|result| < 2^{-149}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ≥ 2^{128}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{               ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFPS (EVEX encoded versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS __m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m128 _mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

XRSTOR—Restore Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /5 XRSTOR *mem*	M	V/V	XSAVE	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F AE /5 XRSTOR64 *mem*	M	V/N.E.	XSAVE	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.8, "Operation of XRSTOR," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED ← RFBM AND RSTORMASK;
        TO_BE_INITIALIZED ← RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
        FI;

        IF RFBM[1] = 1 OR RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
                XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0; // this step does not initialize MXCSR
                XINUSE[1] ← 0;
        FI;

        FOR i ← 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset n from base of XSAVE area;
                        // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
                    XINUSE[i] ← 1;
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN    /* compacted form not supported */
                #GP(0);
```

```
        FI;

        FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
        RESTORE_FEATURES = FORMAT AND RFBM;
        TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
        FORCE_INIT ← RFBM AND NOT FORMAT;
        TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
                XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0;
                MXCSR ← 1F80H;
                XINUSE[1] ← 0;
        FI;

        NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
        FOR i ← 2 TO 62
            IF FORMAT[i] = 1
                THEN
                    IF TO_BE_RESTORED[i] = 1
                        THEN
                            load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                            XINUSE[i] ← 1;
                    FI;
                    NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
            FI;
        ENDFOR;
FI;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
```

```
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void _xrstor(void * , unsigned __int64);

XRSTOR: void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /3 XRSTORS *mem*	M	V/V	XSS	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F C7 /3 XRSTORS64 *mem*	M	V/N.E.	XSS	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.12, "Operation of XRSTORS," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area").
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        initialize x87 state;
        XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        MXCSR ← 1F80H;
        XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
    FI;
ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
```

```
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: void _xrstors(void * , unsigned __int64);

XRSTORS64: void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVE—Save Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /4 XSAVE *mem*	M	V/V	XSAVE	Save state components specified by EDX:EAX to *mem*.
NP REX.W + 0F AE /4 XSAVE64 *mem*	M	V/N.E.	XSAVE	Save state components specified by EDX:EAX to *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.7, "Operation of XSAVE," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
```

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

```
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE: void _xsave(void * , unsigned __int64);

XSAVE: void _xsave64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States with Compaction

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /4 XSAVEC *mem*	M	V/V	XSAVEC	Save state components specified by EDX:EAX to *mem* with compaction.
NP REX.W + 0F C7 /4 XSAVEC64 *mem*	M	V/N.E.	XSAVEC	Save state components specified by EDX:EAX to *mem* with compaction.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.10, "Operation of XSAVEC," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;                /* bitwise logical AND */
TO_BE_SAVED ← RFBM AND XINUSE;          /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
```

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

```
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC: void _xsavec(void * , unsigned __int64);

XSAVEC64: void _xsavec64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /6 XSAVEOPT *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.
NP REX.W + 0F AE /6 XSAVEOPT64 *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.9, "Operation of XSAVEOPT," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
```

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

```
OLD_BV ← XSTATE_BV field from XSAVE header;
TO_BE_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,00000000_00000000H⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void _xsaveopt(void * , unsigned __int64);

XSAVEOPT: void _xsaveopt64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSAVES—Save Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /5 XSAVES *mem*	M	V/V	XSS	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.
NP REX.W + 0F C7 /5 XSAVES64 *mem*	M	V/N.E.	XSS	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.11, "Operation of XSAVES," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
TO_BE_SAVED ← RFBM AND XINUSE;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,COMPMASK⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8        // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] ← 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES: void _xsaves(void * , unsigned __int64);

XSAVES64: void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int _xtest(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0. If LOCK prefix is used.

13.Updates to Chapter 7, Volume 2D

Change bars show changes to Chapter 7of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D:* Instruction Set Reference.

Changes to this chapter:

Updates to the following instructions are covered here with change bars: VRSQRT28SD and VRSQRT28SS.

VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	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

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

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

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SD (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63: 0] ← 0
        FI;
    FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAXVL-1:128] ← 0
```

Table 6-8. VRSQRT28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SD __m128d _mm_rsqrt28_round_sd(__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_mask_rsqrt28_round_sd(__m128d s, __mmask8 m,__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_maskz_rsqrt28_round_sd(__mmask8 m,__m128d a, __m128d b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

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

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

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

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SS (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[31: 0] ← (1.0/ SQRT(SRC[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
```

Table 6-10. VRSQRT28SS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_mask_rsqrt28_round_ss(__m128 s, __mmask8 m,__m128 a,__m128 b, int rounding);
VRSQRT28SS __m128 _mm_maskz_rsqrt28_round_ss(__mmask8 m,__m128 a,__m128 b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

14.Updates to Chapter 8, Volume 3A

Change bars show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Changes to this chapter: Updated Figure 8-6 "Conceptual Six-Level Topology and 32-bit APIC ID Composition" (previously this was a five-level topology). Additional information provided in Section 8.9.4 "Algorithm for Three-Level Mappings of APIC_ID". Updated Example 8-19 "Support Routines for Identifying Package, Core and Logical Processors from 32-bit x2APIC ID". Correction to Example 8-25 "An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop" and Example 8-26 "An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop".

CHAPTER 8
MULTIPLE-PROCESSOR MANAGEMENT

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 recommend 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 it's 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).

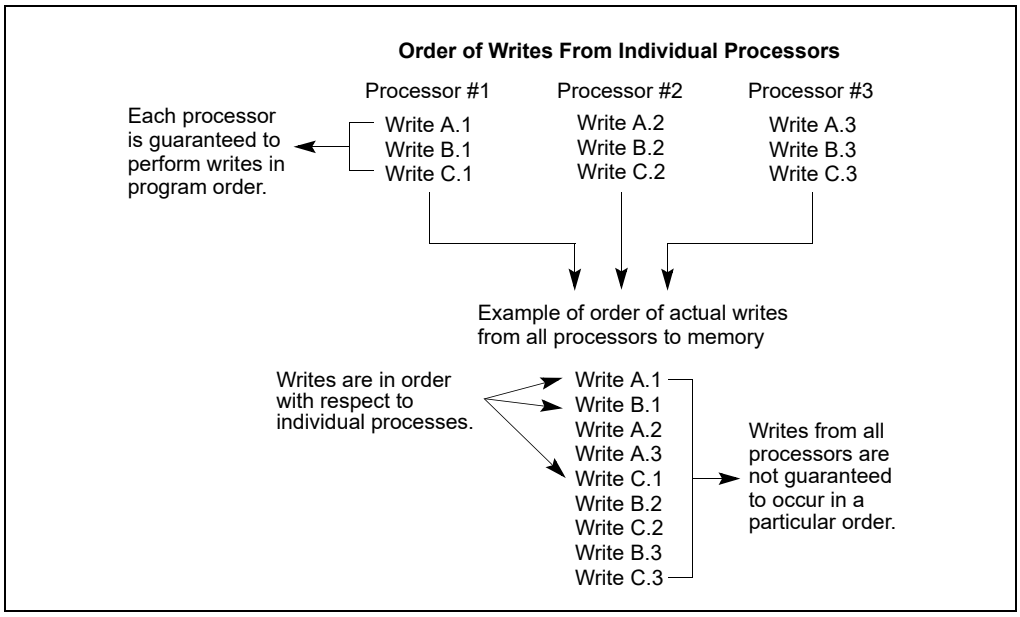


Figure 8-1. Example of Write Ordering in Multiple-Processor Systems

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 MOVS/STOS 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
rep:stosd [_x]	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_x] to 511[_x]= 0, _x <= _y < _z < _x+512 r1 = 1 and r2 = 0 is allowed	

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
rep:stosd [_x] mov ecx, $128 rep:stosd 512[_x]	 mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_x] to 1023[_x]= 0, _x <= _y < _x+512 < _z < _x+1024 r1 = 1 and r2 = 0 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 ready 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
rep:stosd [_x] mov [_z], $1	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 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 CR8[3]), MOV (to debug register), WBINVD, and WRMSR[4].
- **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.
- 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.")

---

3. MOV CR8 is not defined architecturally as a serializing instruction.
4. WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.
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.

- 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 summarizes 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, FFFFFF00H; 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 process 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.

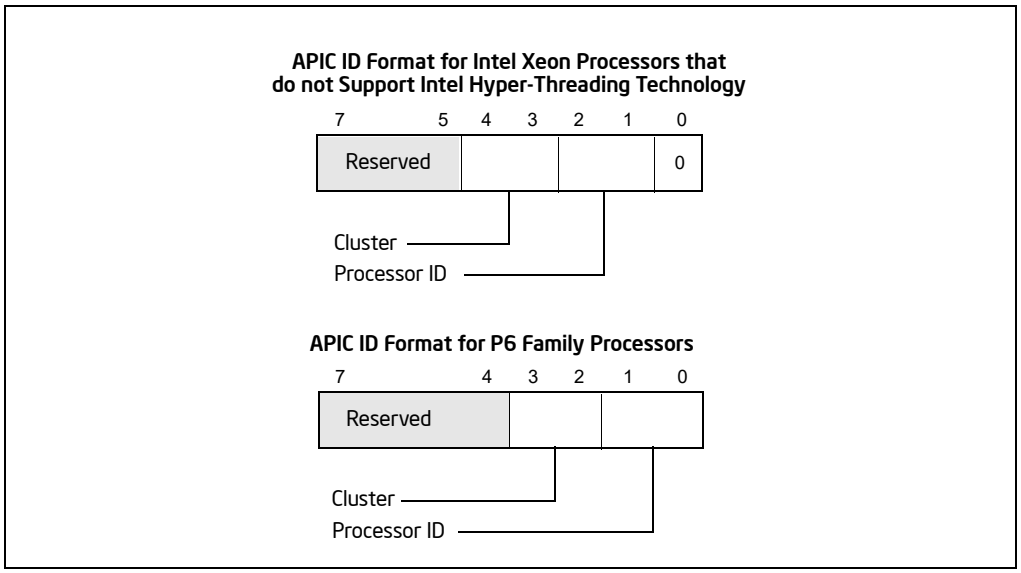


Figure 8-2. Interpretation of APIC ID in Early MP Systems

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 represents the hardware capability of the physical processor.[6]

- **Addressable IDs for processor cores in the same Package[7] (CPUID.(EAX=4, ECX=0[8]):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 or 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.

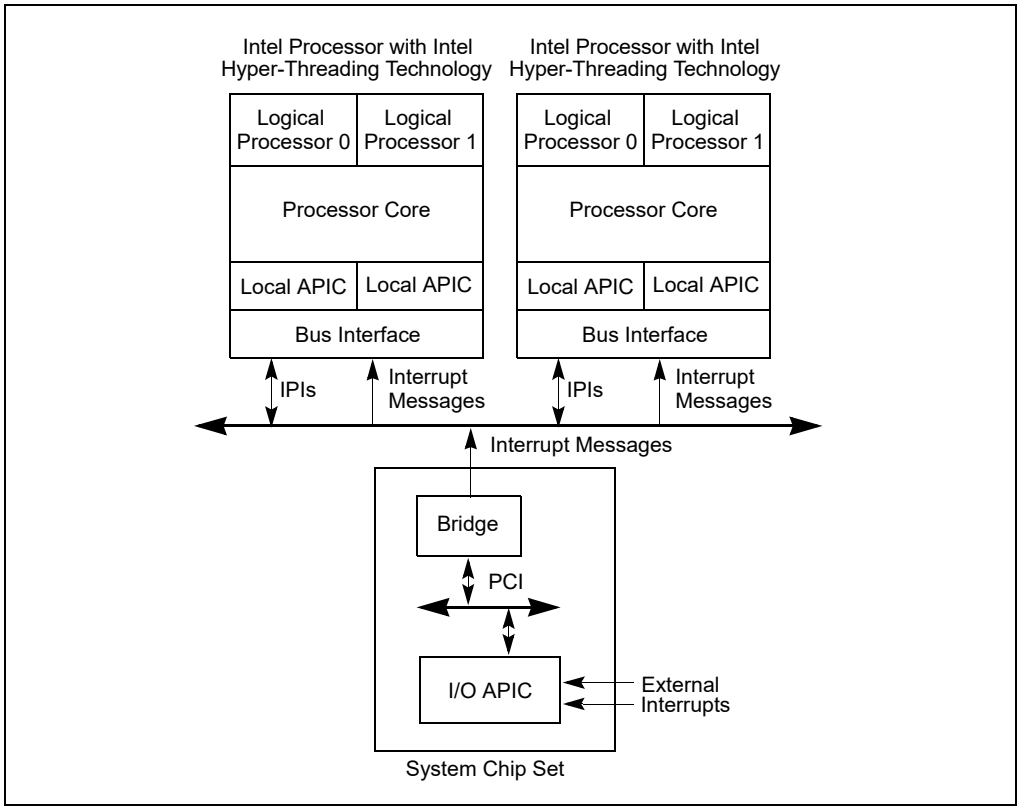


Figure 8-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology

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).

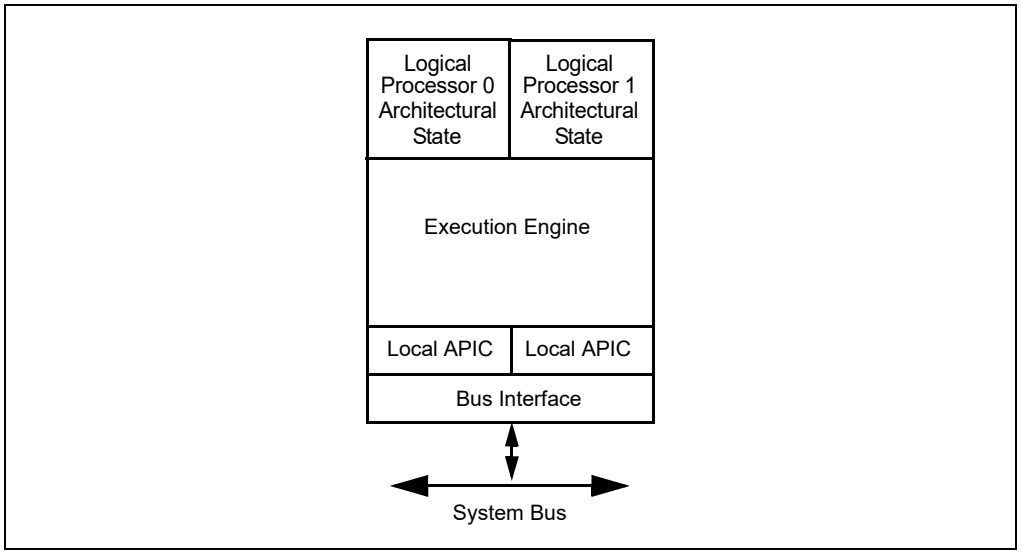


Figure 8-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology

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) MSRs (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) MSRs (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 14.7, "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. If the processor does not support CPUID leaf 0BH, the 8-bit initial APIC ID can represent 4 levels of hierarchy:

- **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 to three levels of hierarchy.
- **Package** — A multi-processor system consists of two or more sockets, each mates with a physical processor package. The PACKAGE_ID sub-field distinguishes different physical packages within a cluster.
- **Core** — A physical processor package consists of one or more processor cores. The CORE_ID sub-field distinguishes processor cores in a package. 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).

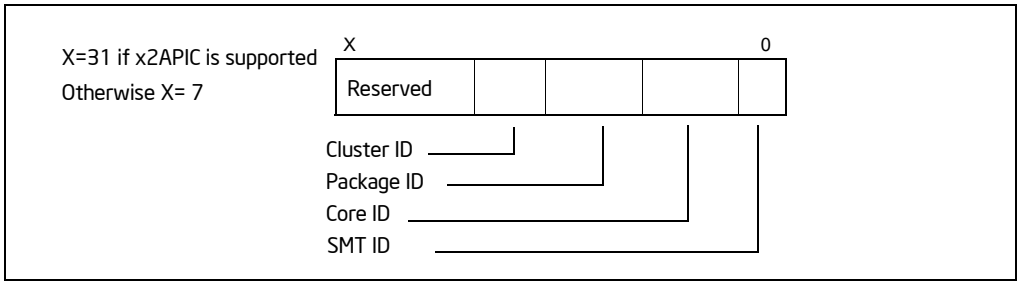


Figure 8-5. Generalized Four level Interpretation of the APIC ID

If the processor supports CPUID leaf 0BH, 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. Common processor families may employ topology similar to that represented by 8-bit Initial APIC ID. In general, CPUID leaf 0BH can support topology enumeration algorithm that decompose a 32-bit APIC ID into more than four sub-fields (see Figure 8-6).

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 8-20).

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.

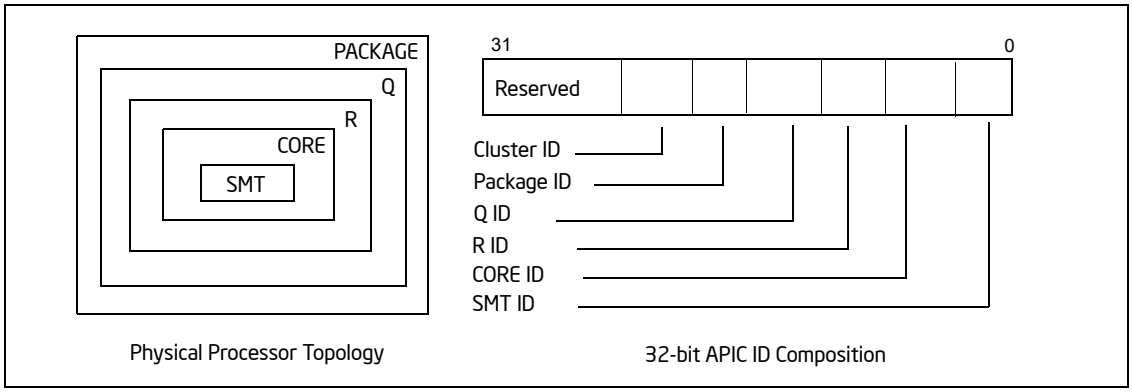


Figure 8-6. Conceptual Six-Level Topology and 32-bit APIC ID Composition

8.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf

CPUID leaf 0BH provides 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. 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, 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, ECX=3):ECX[15:8] will report 00H, indicating sub leaf 03H 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; // query each sub leaf of CPUID leaf 0BH
        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, and ECX =0 is specified as input when executing CPUID, CPUID.(EAX=0BH, 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, 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, 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];
```

Currently, only the following encoding of hierarchical level type are defined: 0 (invalid), 1 (SMT), and 2 (core). Software must not assume any "level type" encoding value to be related to any sub-leaf index, except sub-leaf 0.

Example 8-16 and Example 8-17 represent the general technique for using CPUID leaf 0BH to enumerate processor topology of more than two levels of hierarchy inside a physical package. Most processor families to date requires only "SMT" and "CORE" levels within a physical package. The examples in later sections will focus on these three-level topology only.

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 affinitizing-current-thread-context (See Example 8-22) 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.

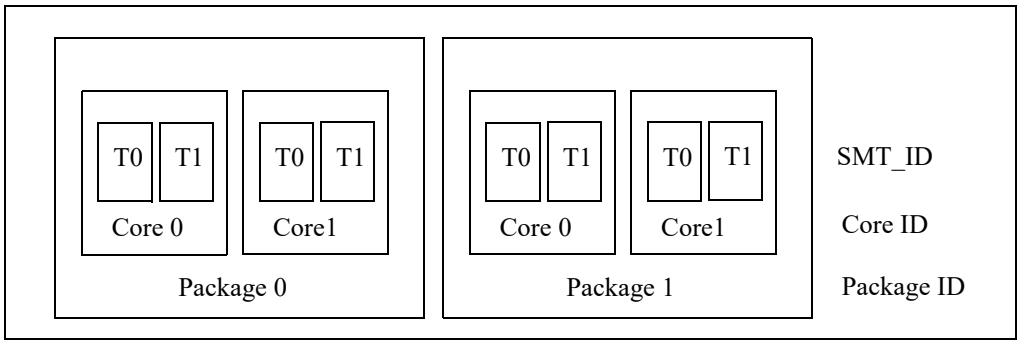


Figure 8-7. Topological Relationships between Hierarchical IDs in a Hypothetical MP Platform

Table 8-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology[1]

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 runtime[9] 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.4[10]:EAX[31:26], and CPUID.4[11]: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 8-18, 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 8-18. 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 8-19. 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 8-20. 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 8-20. 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 8-21 and support routines shown in Example 8-18):

- 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 8-22a.
- 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 8-22b. 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 8-22. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same core.

In Example 8-21, 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 8-21. 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 8-19);
    // otherwise, SMT_MASK is determined using CPUID leaf 01H and leaf 04H (Example 8-20).
    // 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 8-19 or Example 8-20
    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 8-19 or Example 8-20
    PACKAGE_ID = (APIC_ID & PACKAGE_MASK) ;
}
```

Example 8-22. 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 8-19 or 8-bit ID in Example 8-20
                Extract the Package_ID, Core_ID and SMT_ID as explained in three level extraction
                      algorithm of Example 8-21
                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;
```

```
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 PACKGE 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 improves 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 de-pipelines 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 8-23.

Example 8-23. 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 8-24:

Example 8-24. 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 >= IdleTimeThreshhold) 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 8-25. 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 >= IdleTimeThreshhold) 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 8-26:

Example 8-26. 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 >= IdleTimeThreshhold) 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.

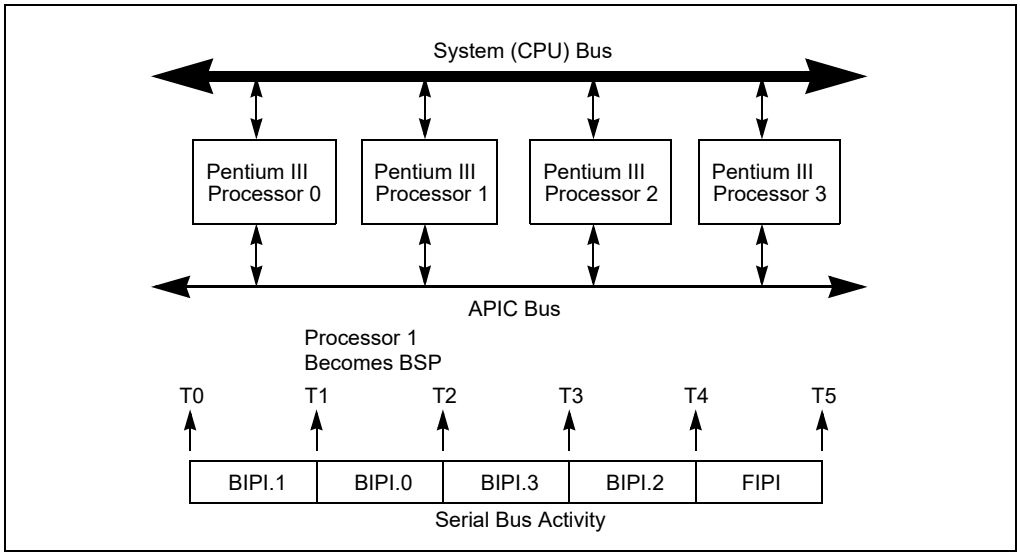


Figure 8-1. MP System With Multiple Pentium III Processors

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.

15. Updates to Chapter 9, Volume 3A

Change bars 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: Minor update to Section 9.8.5 "Initializing IA-32e Mode".

CHAPTER 9
PROCESSOR MANAGEMENT AND INITIALIZATION

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 is in real-address mode with paging disabled.

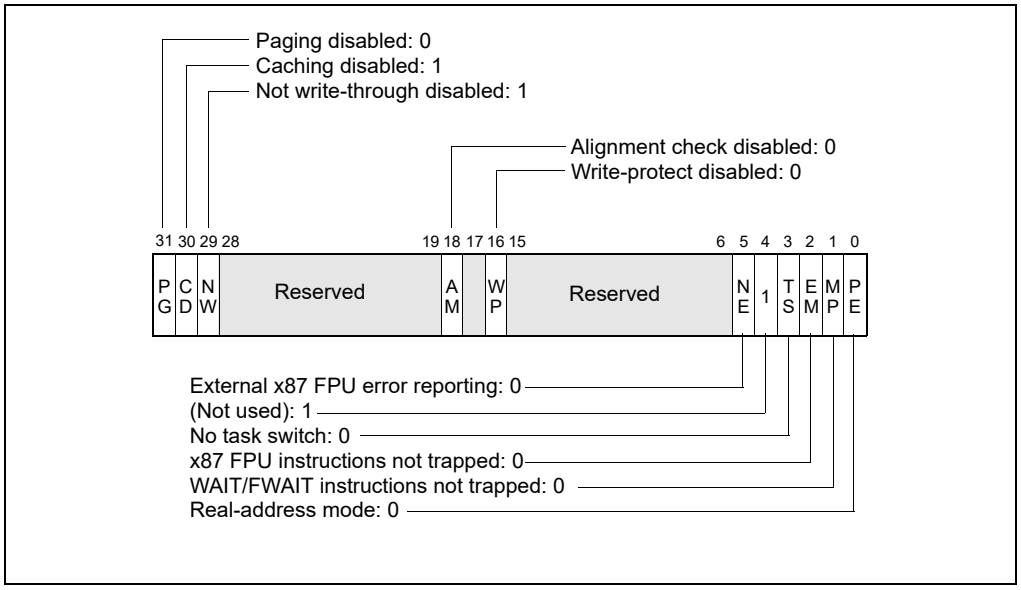


Figure 9-1. Contents of CR0 Register after Reset

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
XCR0	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.[11] = 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).

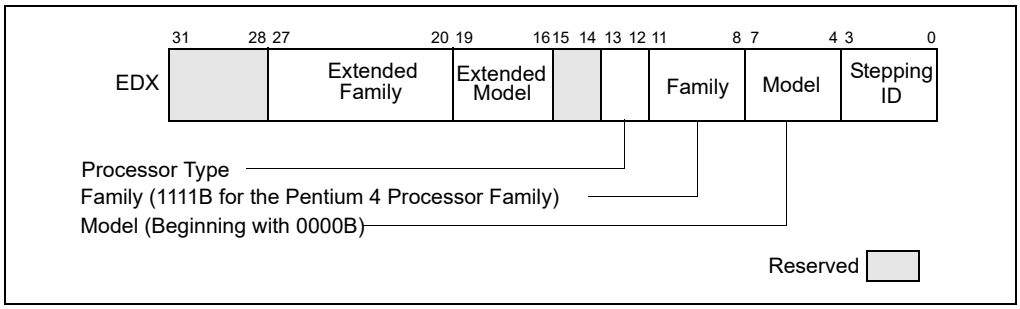


Figure 9-2. Version Information in the EDX Register after Reset

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

CR0 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 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 levels of enhanced paging structures (see Section 4.5, “4-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).
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 tables 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.

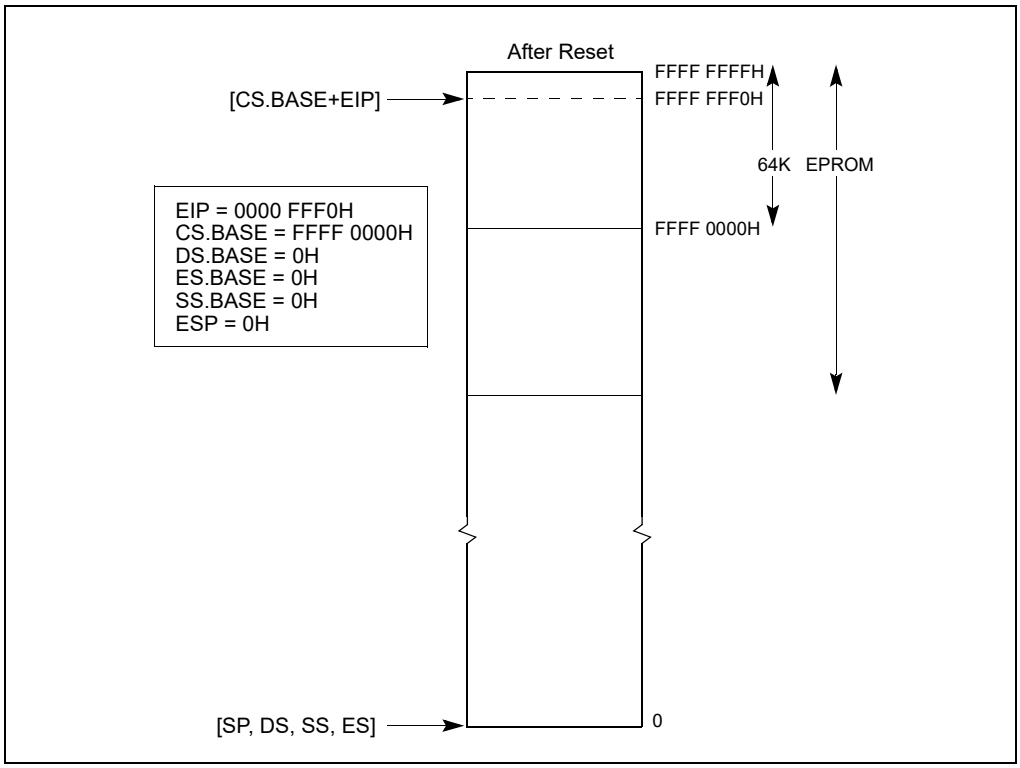


Figure 9-3. Processor State After Reset

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 CR0 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 ?
```

```
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_regDD ?
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_baseDW ?
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_limDW ?
98      table_linearDD ?
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
```

```
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 MOVS    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 MOVS    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
```

```
 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.
```

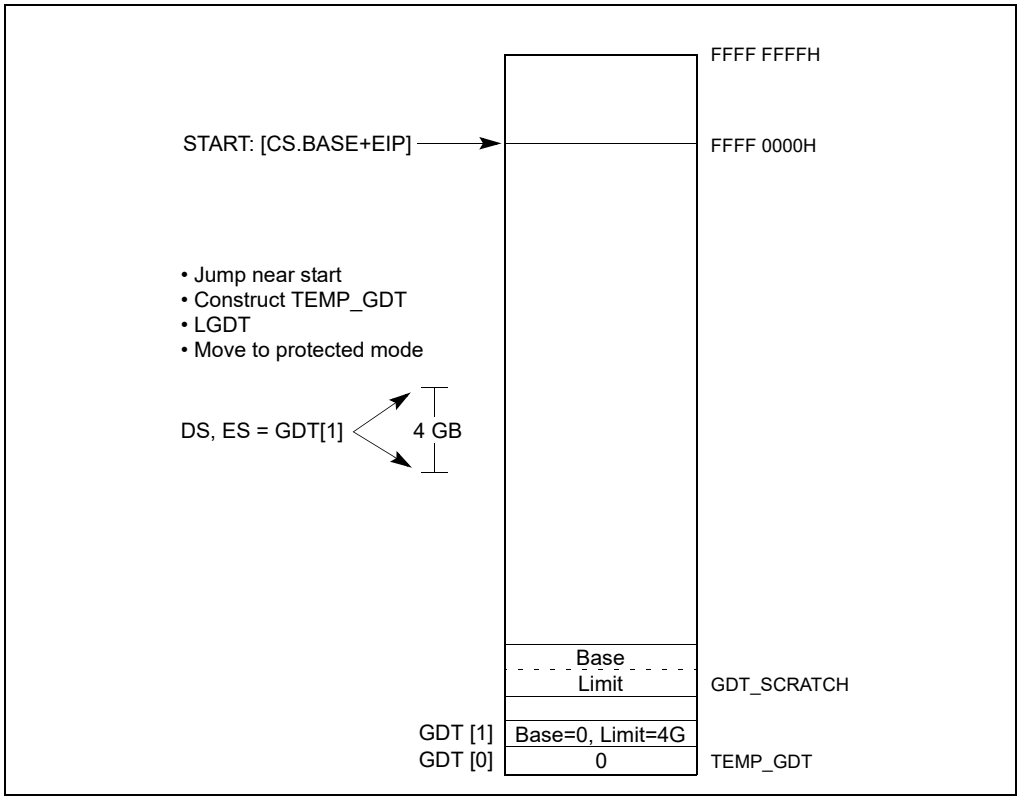


Figure 9-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

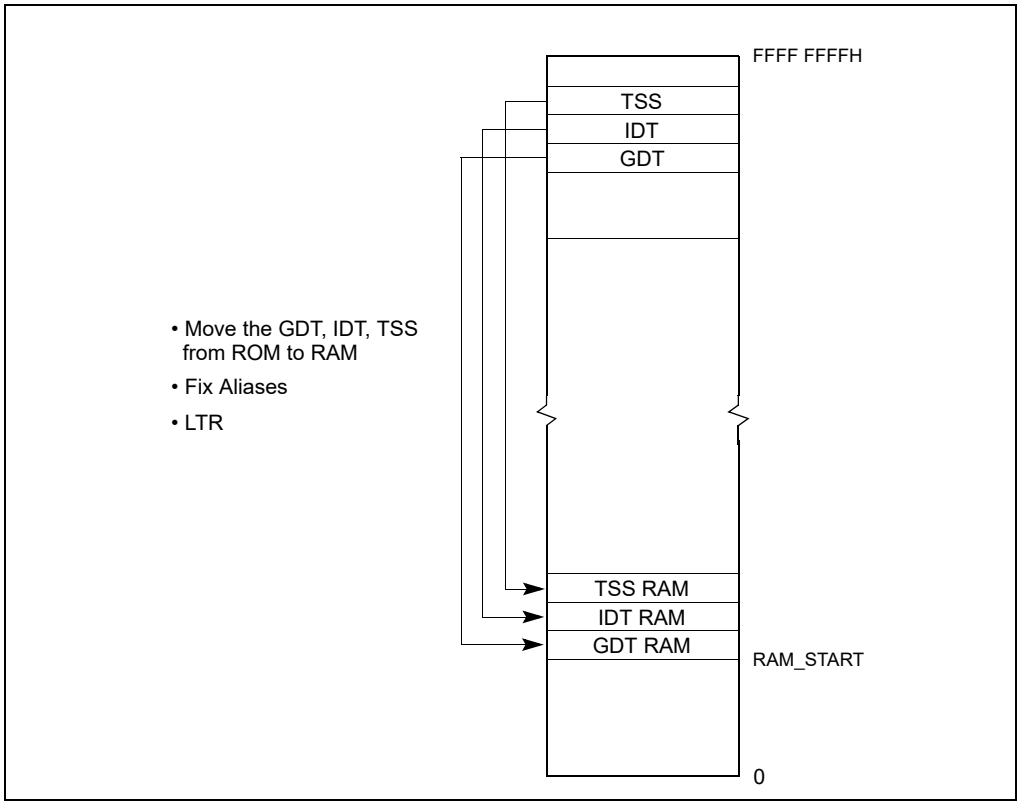


Figure 9-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

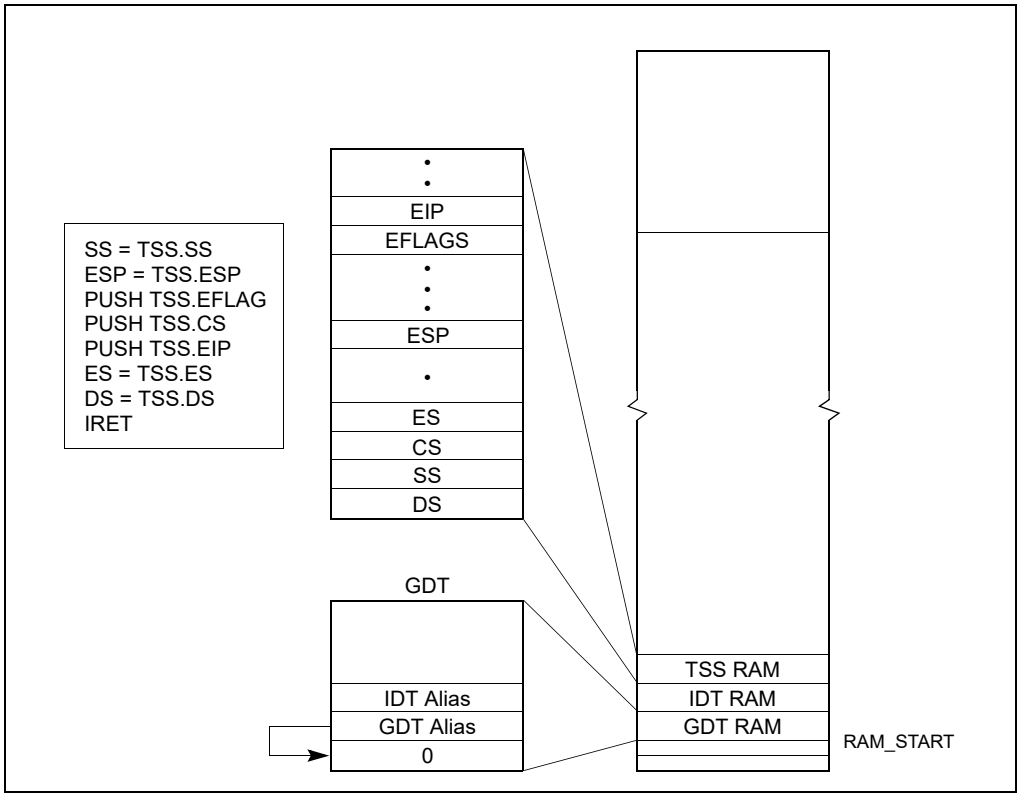


Figure 9-6. Task Switching (Lines 282-296 of List File)

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..0FFFFFFFFH))
                   -- 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 0FFFFFFF0H 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 = 0FFFF0000H) ...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.

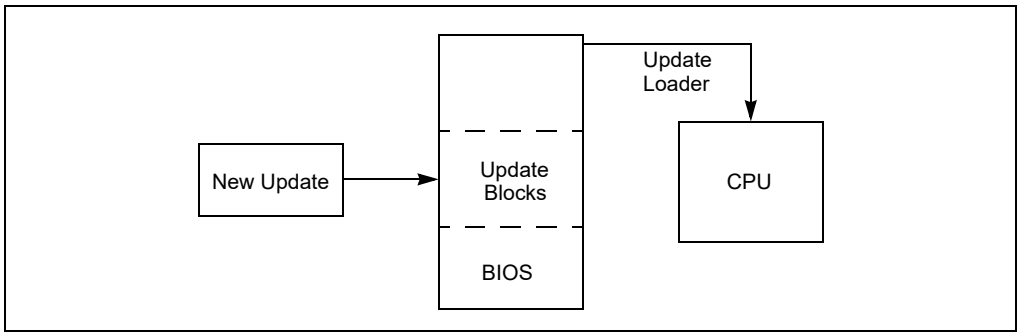


Figure 9-7. Applying Microcode Updates

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 micro-code 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	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* 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	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* 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.
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	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. 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.

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. 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 steppings.

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 0FFFFFFFFH 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 DO42 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.

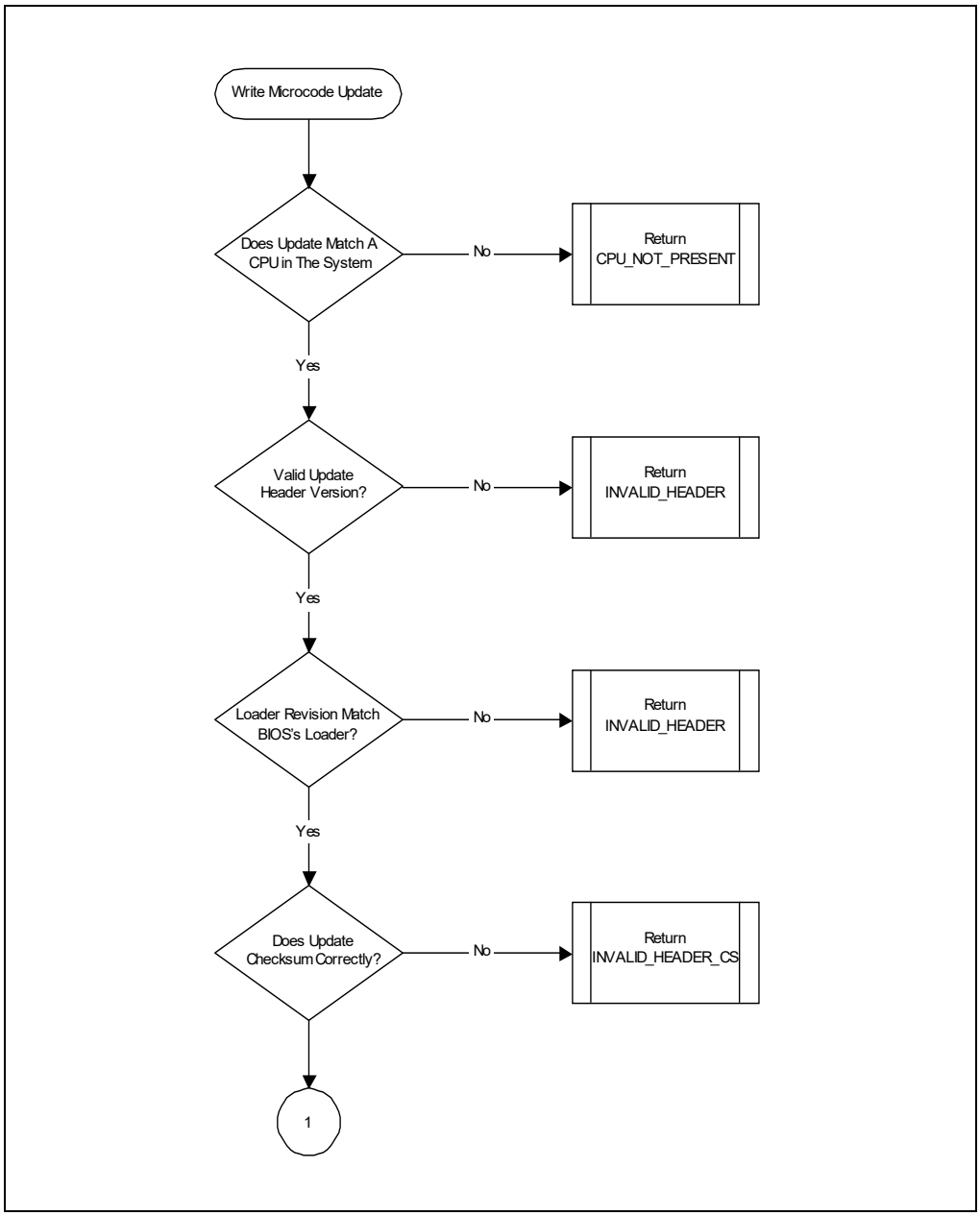


Figure 9-8. Microcode Update Write Operation Flow [1]

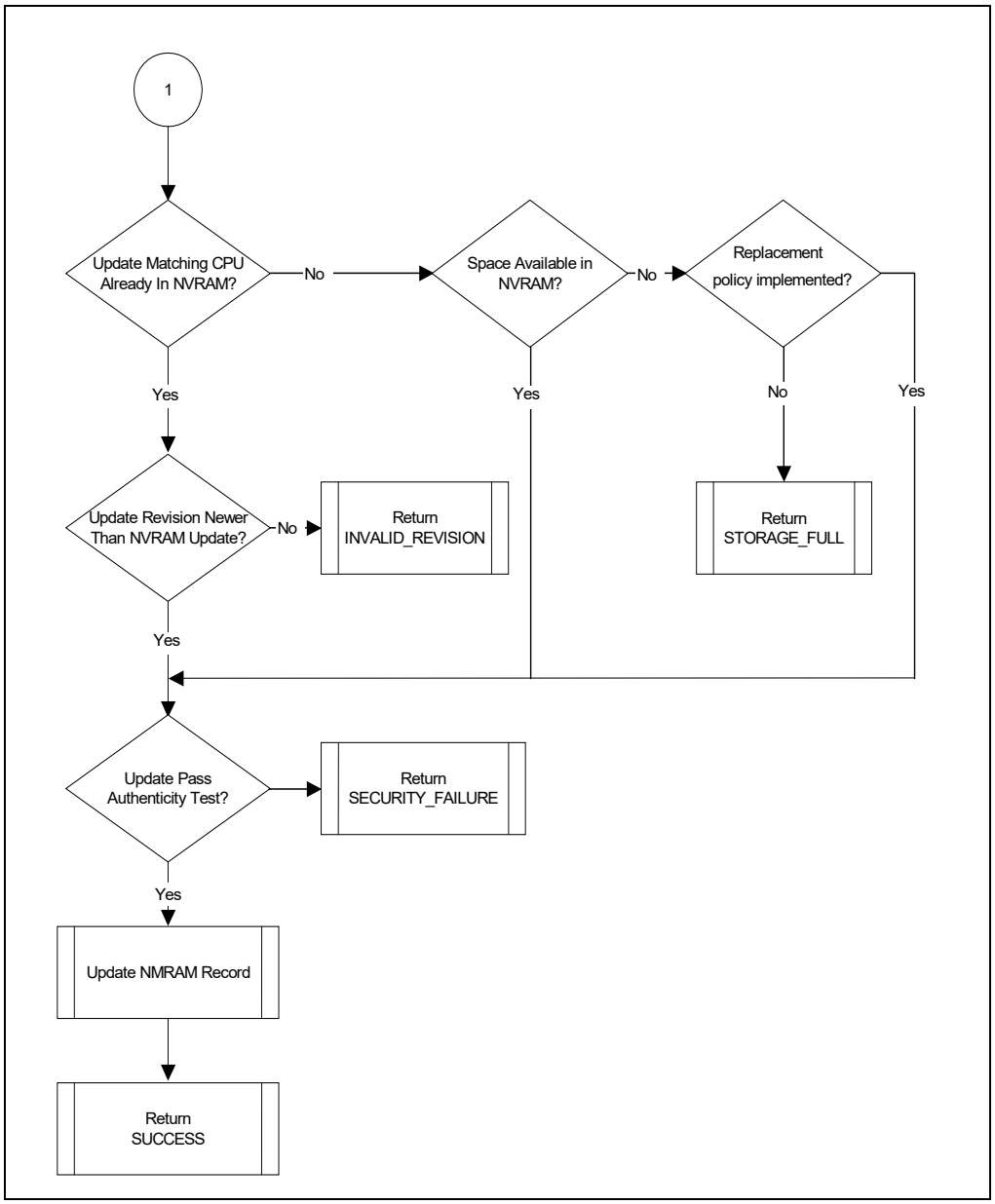


Figure 9-9. Microcode Update Write Operation Flow [2]

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 0FFFFFFFFH 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 0FFFFFFFFH. 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.

16.Updates to Chapter 10, Volume 3A

Change bars show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Changes to this chapter: Update to Section 10.12.3 "MSR Access in x2APIC Mode".

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 14.7.2, "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.

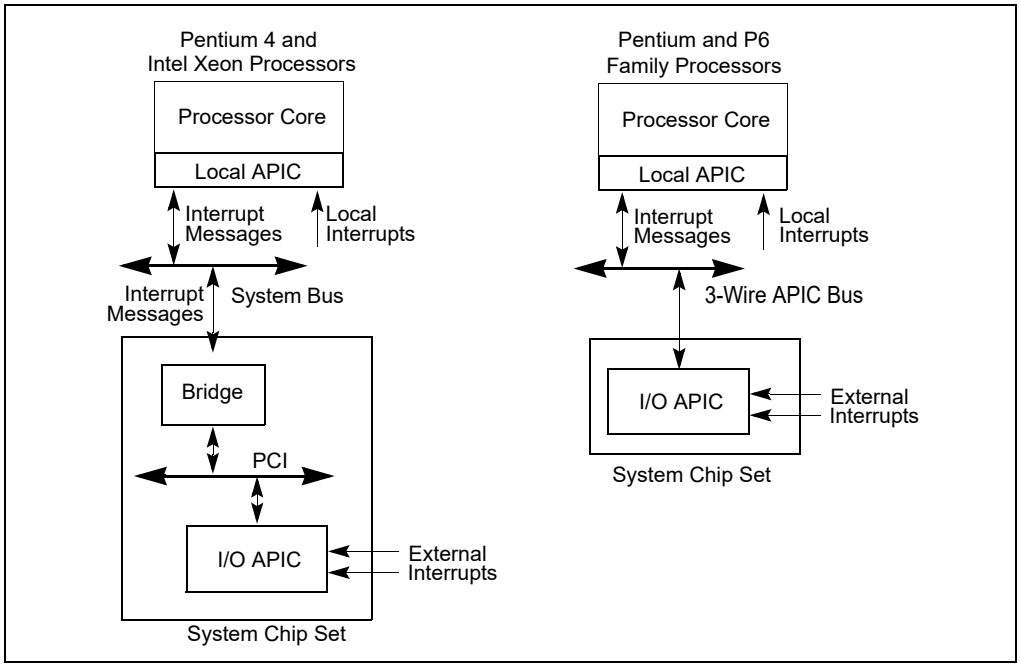


Figure 10-1. Relationship of Local APIC and I/O APIC In Single-Processor Systems

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.

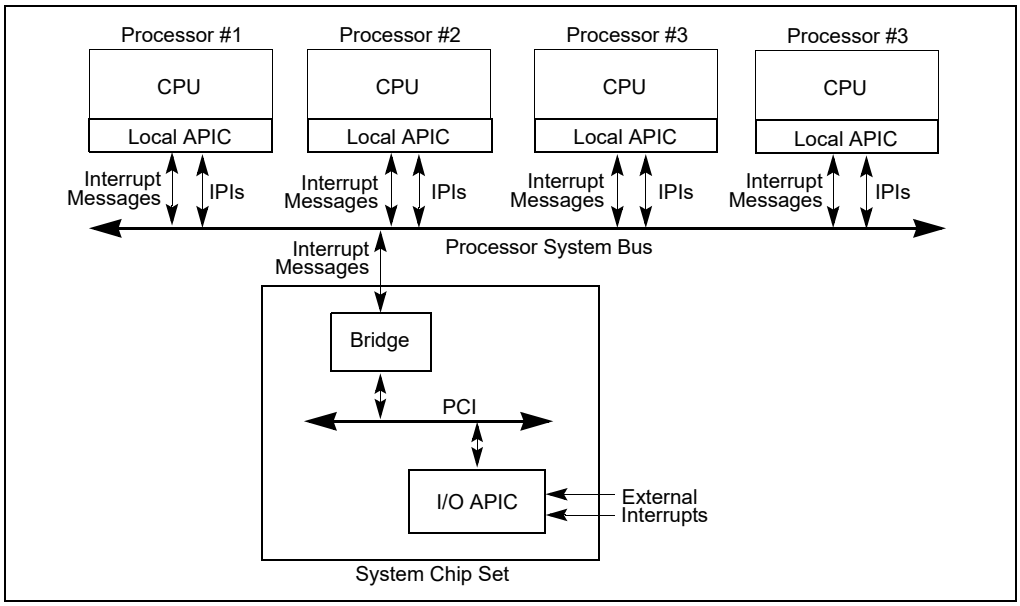


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

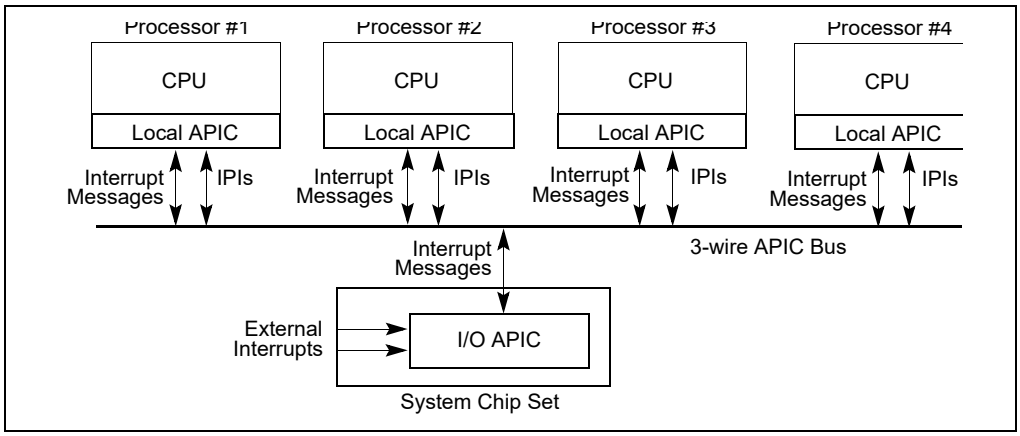


Figure 10-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems

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 communication 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.

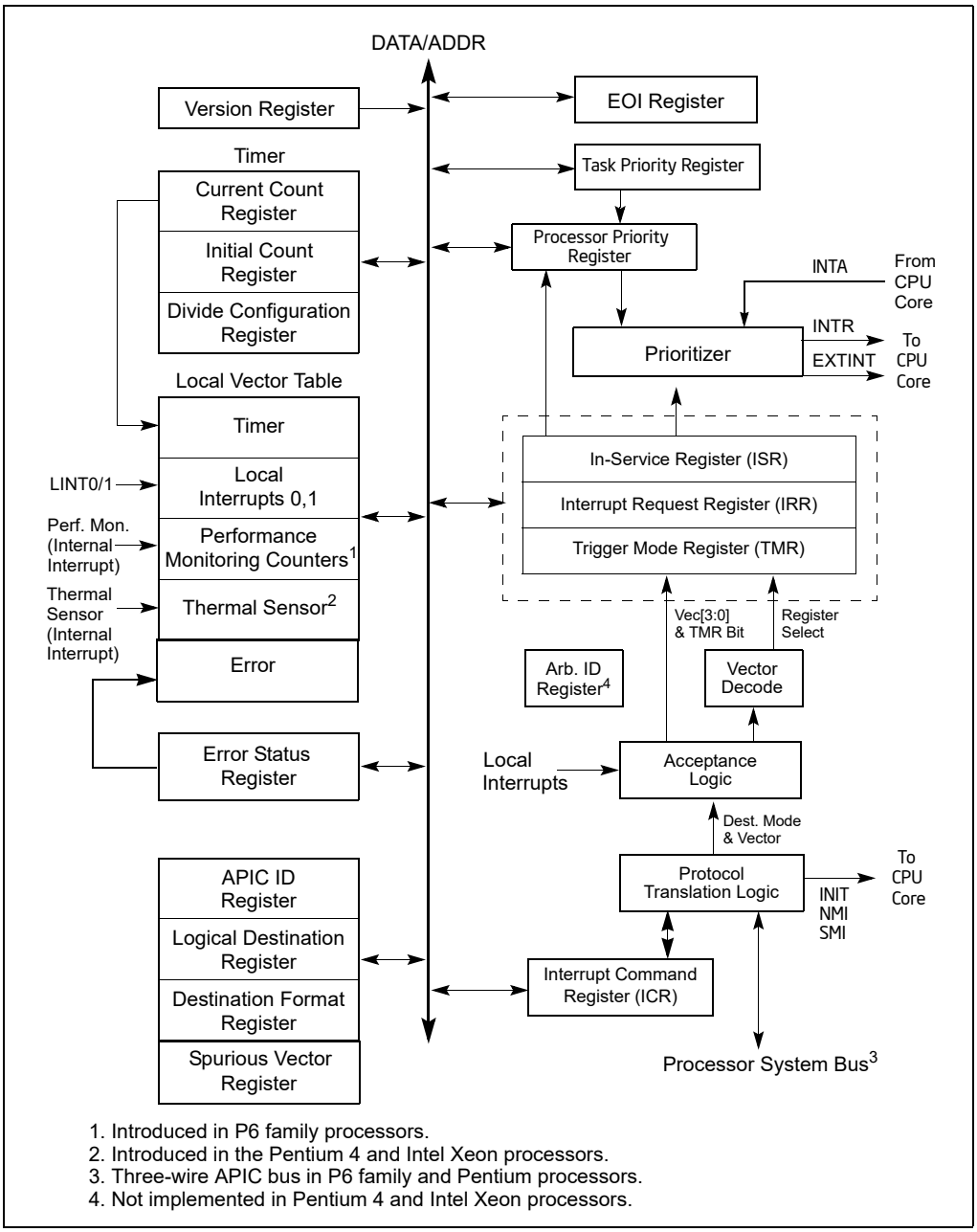


Figure 10-4. Local APIC Structure

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 Nehalem[1] 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 2.7, "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:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 10-5):
   - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 10.4.2, "Presence of the Local APIC") is also set to 0.
   - When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).
   - For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.
   - When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 10.4.7.1, "Local APIC State After Power-Up or Reset."
2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23):
   - If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23). The state of the local APIC when in this software-disabled state is described in Section 10.4.7.2, "Local APIC State After It Has Been Software Disabled."
   - When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

10.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 10-5). MSR bit functions are described below:

- **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 8.4, "Multiple-Processor (MP) Initialization." Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 10.4.3, "Enabling or Disabling the Local APIC"). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.
- **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or reset, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR[2] through 63 in the IA32_APIC_BASE MSR are reserved.

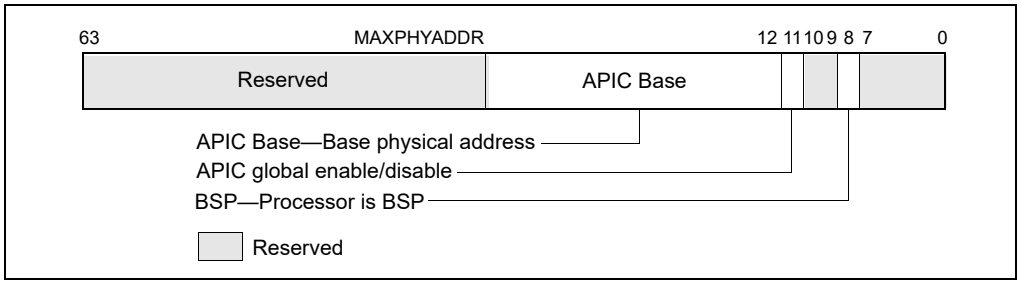


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

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.

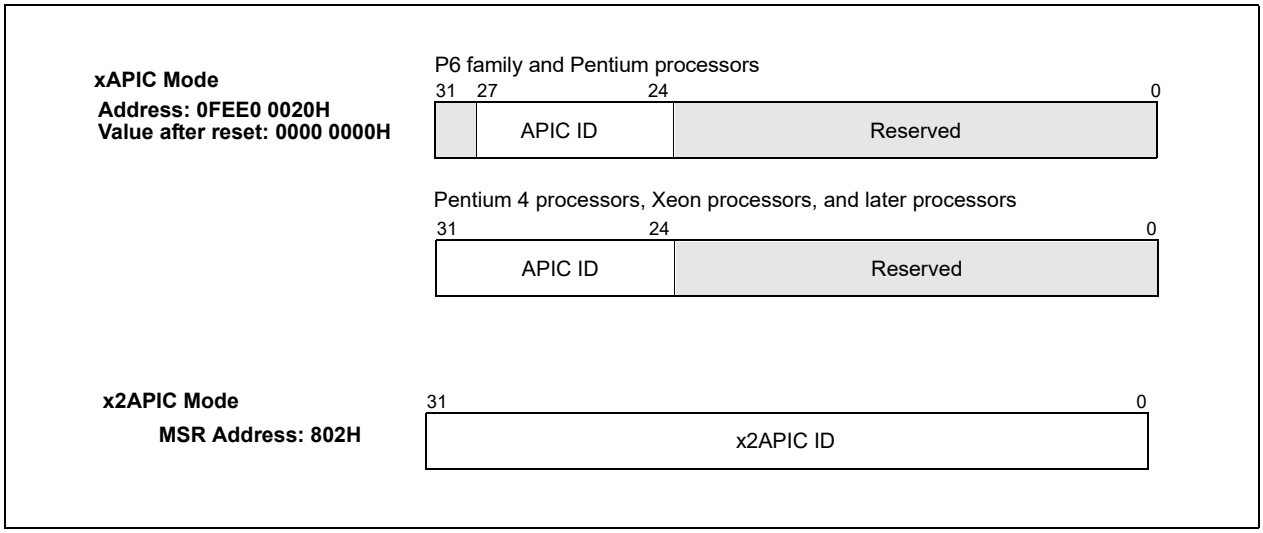


Figure 10-6. Local APIC ID Register

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 affect 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.

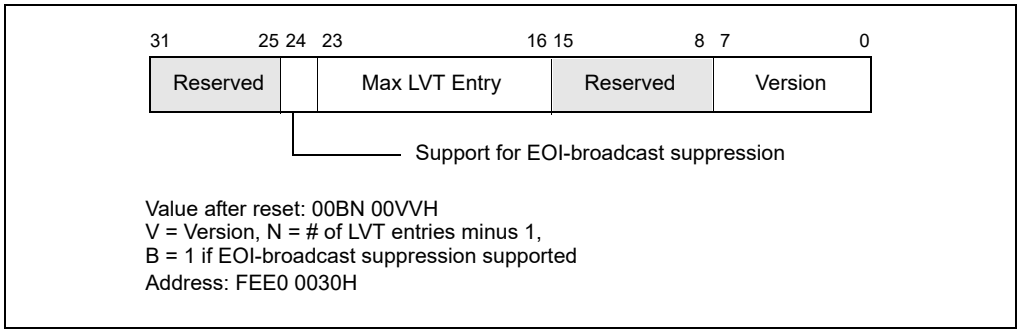


Figure 10-7. Local APIC Version Register

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 14.7.2, "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.

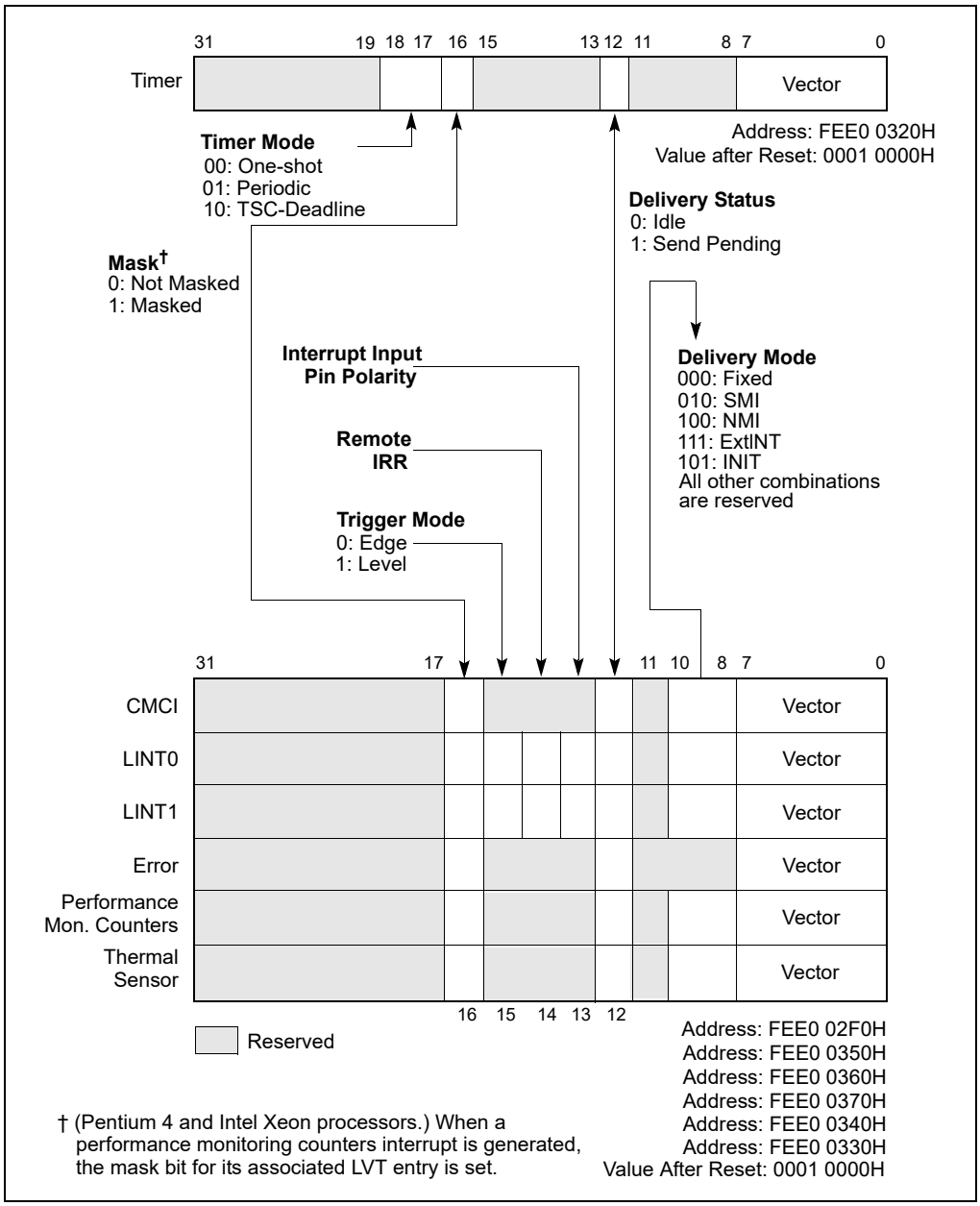


Figure 10-8. Local Vector Table (LVT)

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:

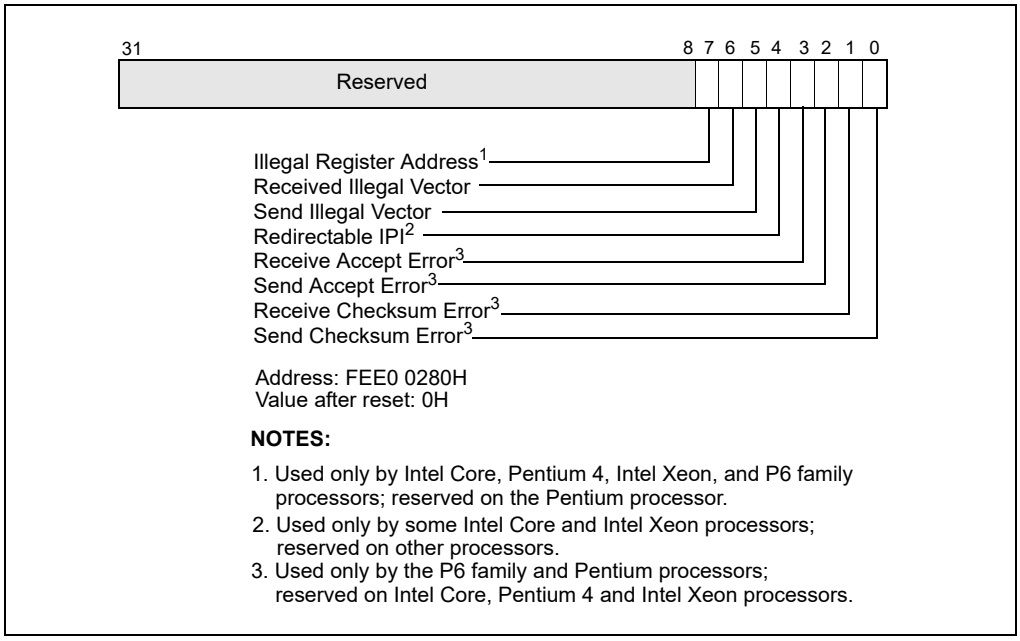


Figure 10-9. Error Status Register (ESR)

- 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.

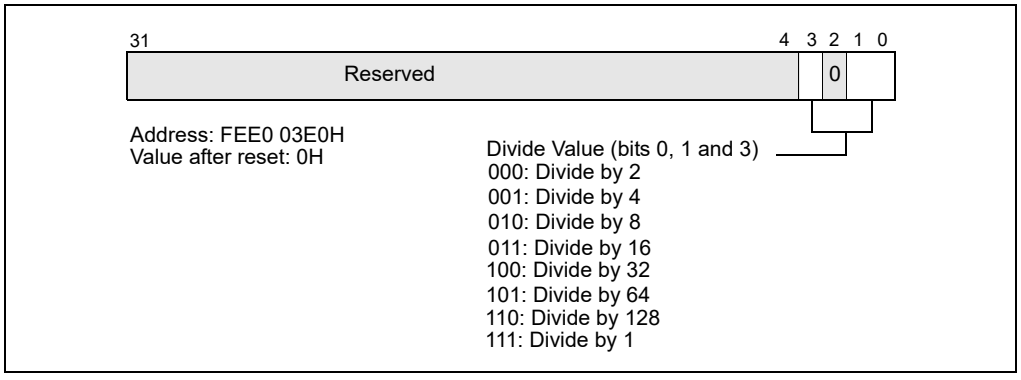


Figure 10-10. Divide Configuration Register

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.

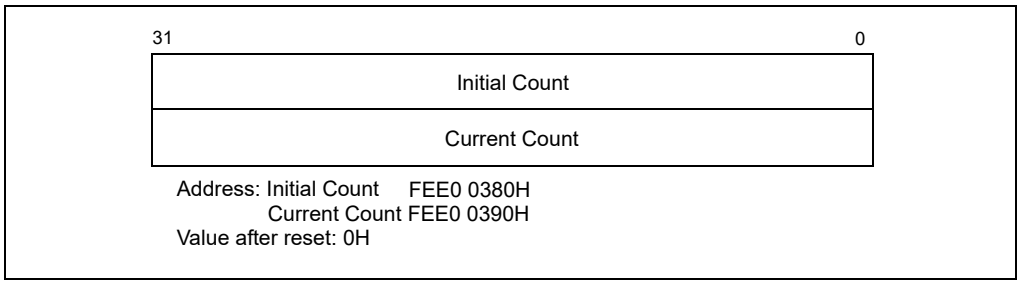


Figure 10-11. Initial Count and Current Count Registers

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, an 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.)

A write to the LVT Timer Register that changes the timer mode disarms the local APIC timer. The supported timer modes are given in Table 10-2. The three modes of the local APIC timer are mutually exclusive.

Table 10-2. Local APIC Timer Modes

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

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-bit[5] 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.

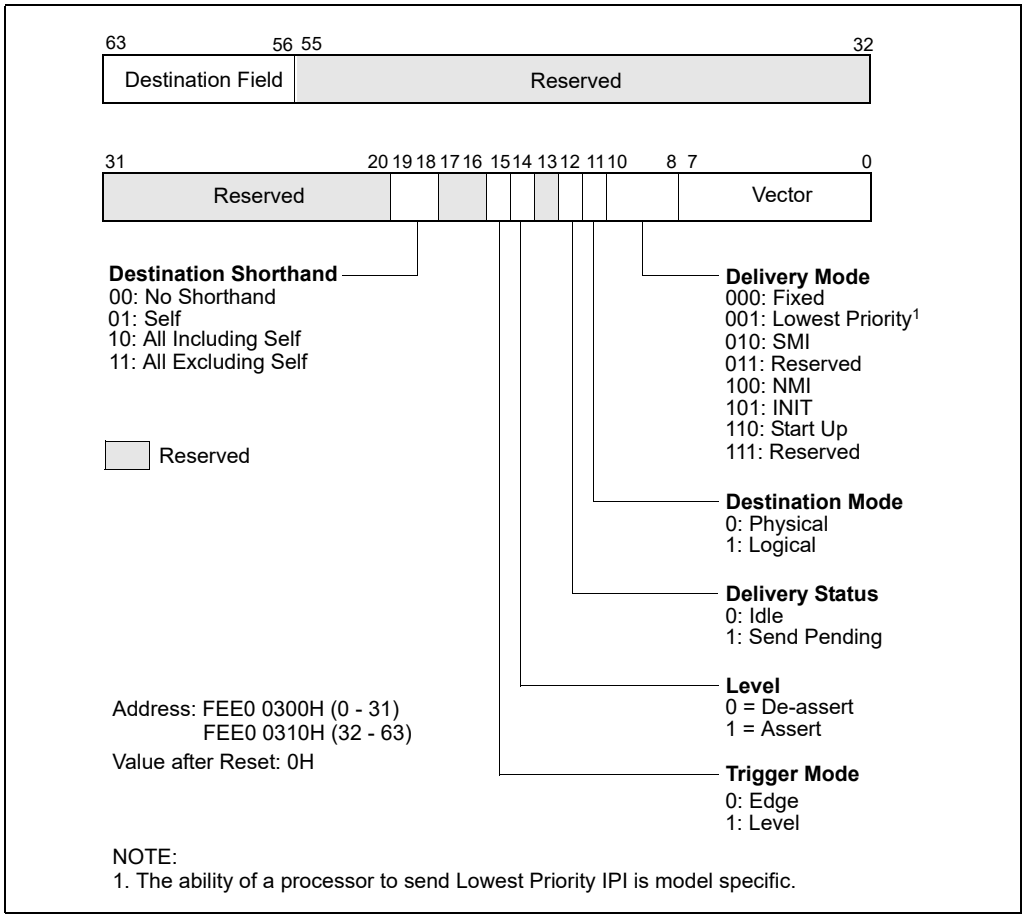


Figure 10-12. Interrupt Command Register (ICR)

5. In XAPIC mode the ICR is addressed as two 32-bit registers, ICR_LOW (FFE0 0300H) and ICR_HIGH (FFE0 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 Specifies the type of IPI to be sent. This field is also know as the IPI message type field.

- **000 (Fixed)** Delivers the interrupt specified in the vector field to the target processor or processors.
- **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.
- **010 (SMI)** Delivers an SMI interrupt to the target processor or processors. The vector field must be programmed to 00H for future compatibility.
- **011 (Reserved)**
- **100 (NMI)** Delivers an NMI interrupt to the target processor or processors. The vector information is ignored.
- **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.
- **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.
- **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.

Destination Mode Selects either physical (0) or logical (1) destination mode (see Section 10.6.2, "Determining IPI Destination").

Delivery Status (Read Only) Indicates the IPI delivery status, as follows:

- **0 (Idle)** Indicates that this local APIC has completed sending any previous IPIs.
- **1 (Send Pending)** Indicates that this local APIC has not completed sending the last IPI.

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 IPI[6] 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.

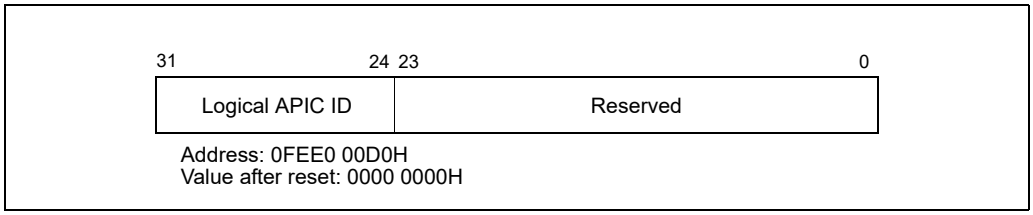


Figure 10-13. Logical Destination Register (LDR)

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.

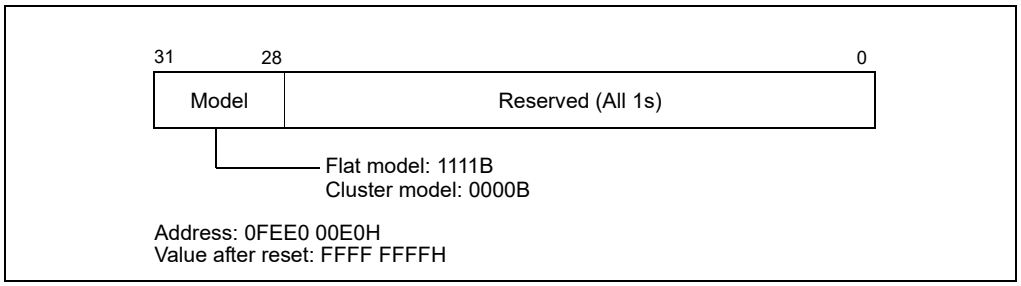


Figure 10-14. Destination Format Register (DFR)

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.

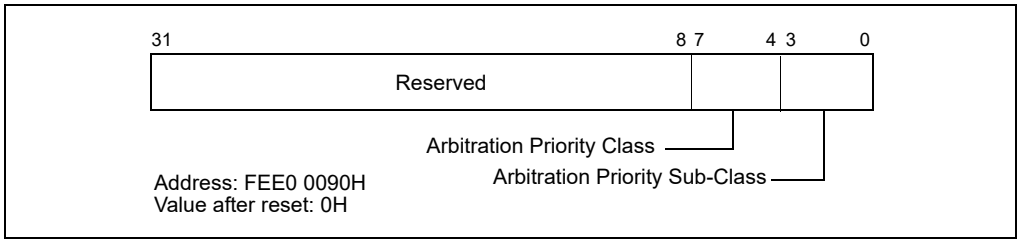


Figure 10-15. Arbitration Priority Register (APR)

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 and 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.

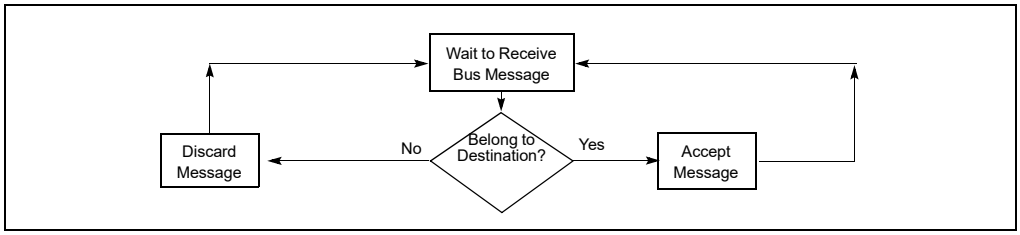


Figure 10-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors)

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 determines 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)

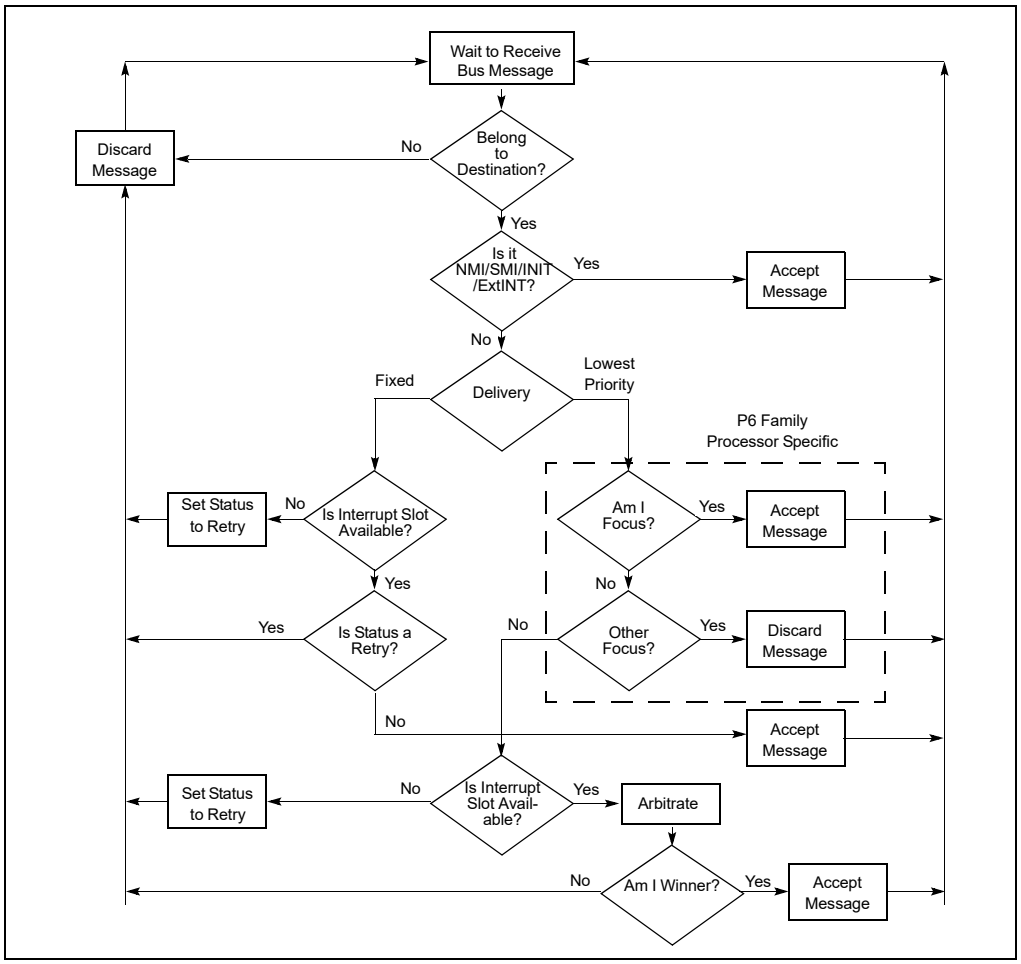


Figure 10-17. Interrupt Acceptance Flow Chart for the Local APIC (P6 Family and Pentium Processors)

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.

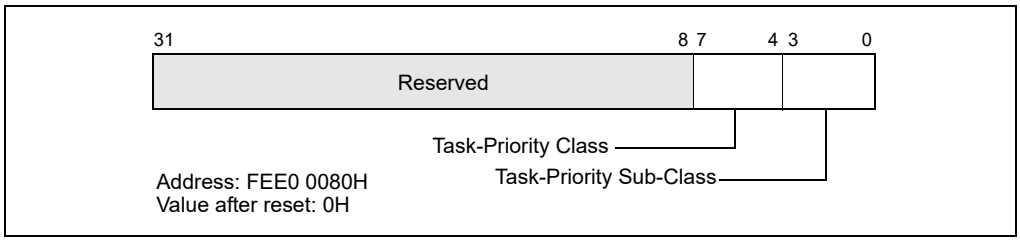


Figure 10-18. Task-Priority Register (TPR)

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.

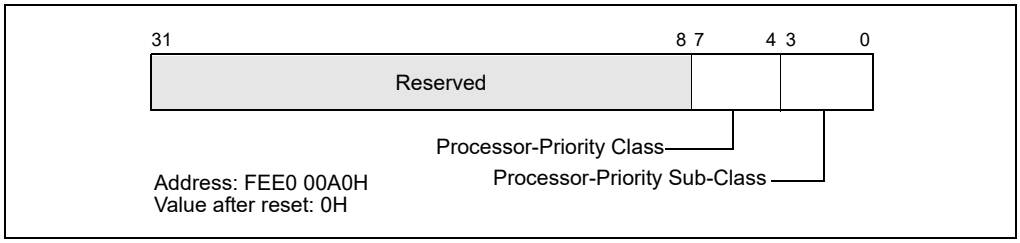


Figure 10-19. Processor-Priority Register (PPR)

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.

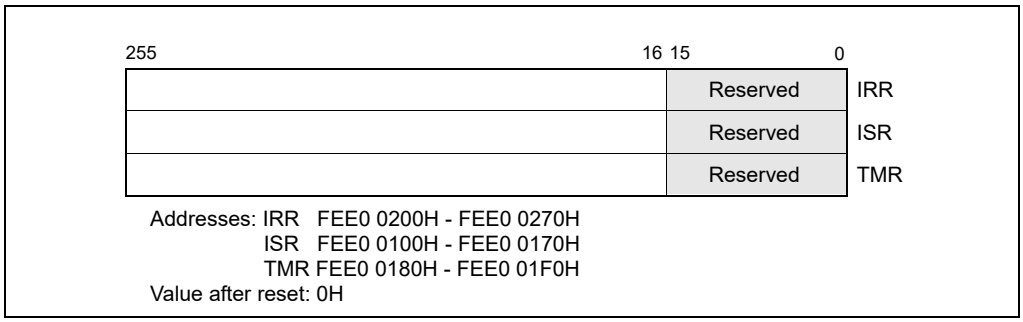


Figure 10-20. IRR, ISR and TMR Registers

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.

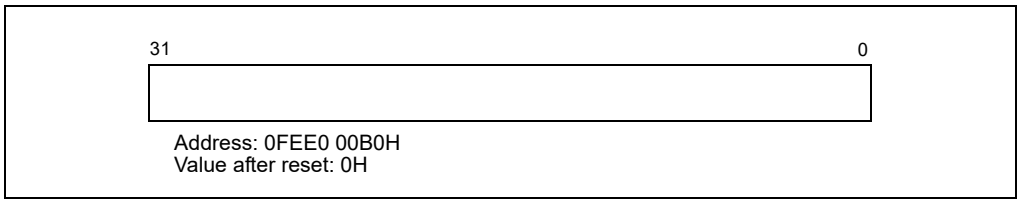


Figure 10-21. EOI Register

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.

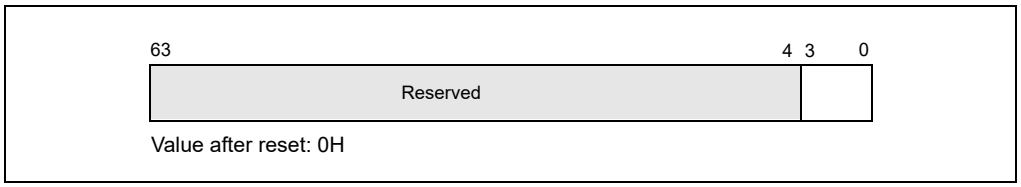


Figure 10-22. CR8 Register

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

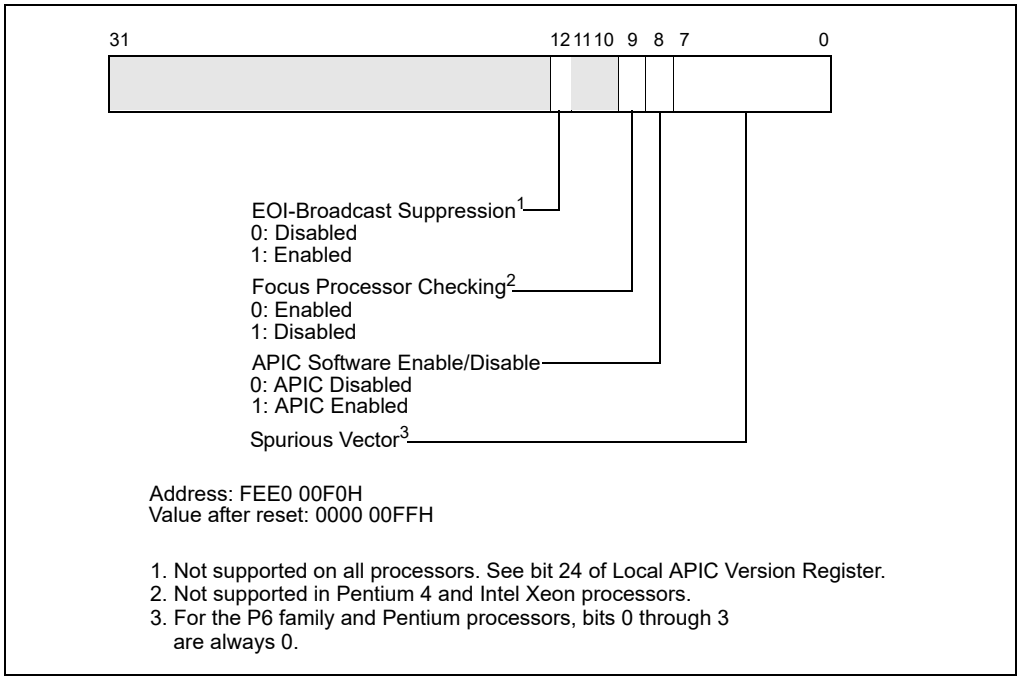


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

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.

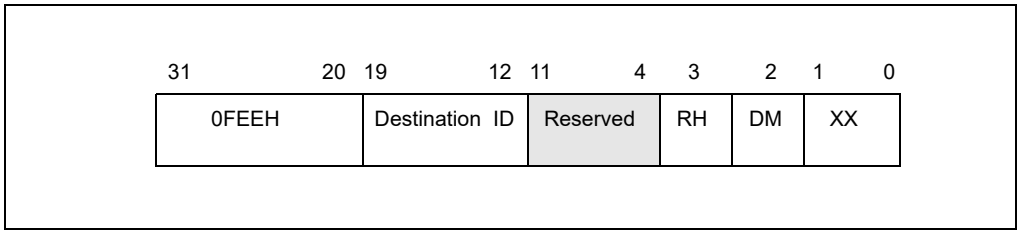


Figure 10-24. Layout of the MSI Message Address Register

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 to 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 re-direction).
    - 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.

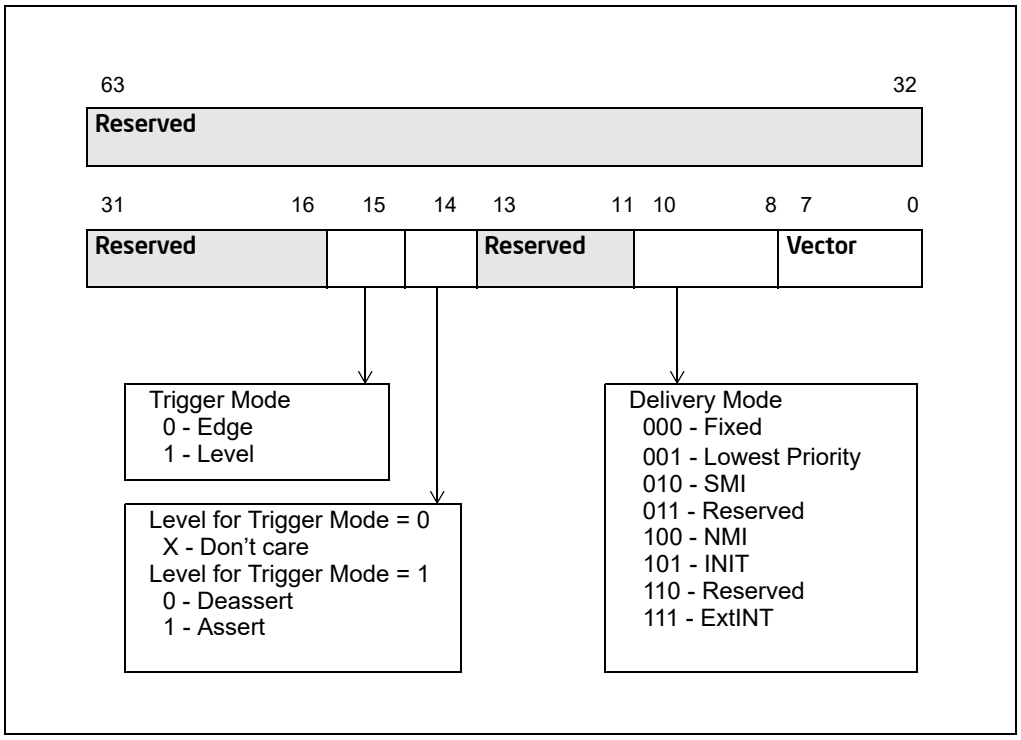


Figure 10-25. Layout of the MSI Message Data Register

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.

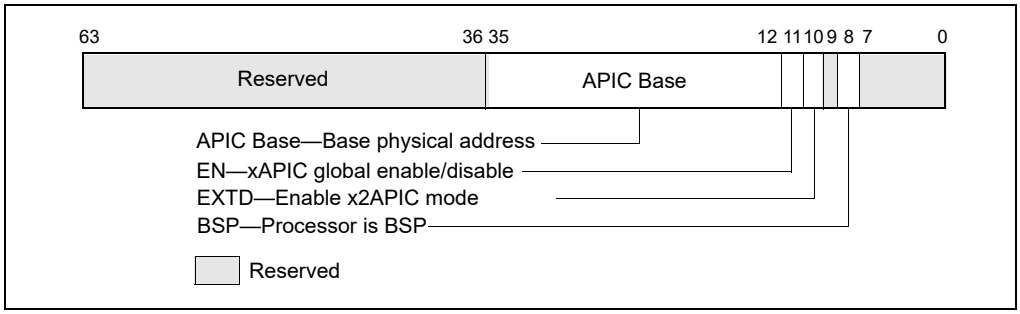


Figure 10-26. IA32_APIC_BASE MSR Supporting x2APIC

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, EXTD = 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, EXTD = 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 BFFH 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–BFFH 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 an 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 "unconnected" 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 0BFFH 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 an 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 & FFFFF800H = 00000800H". 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.

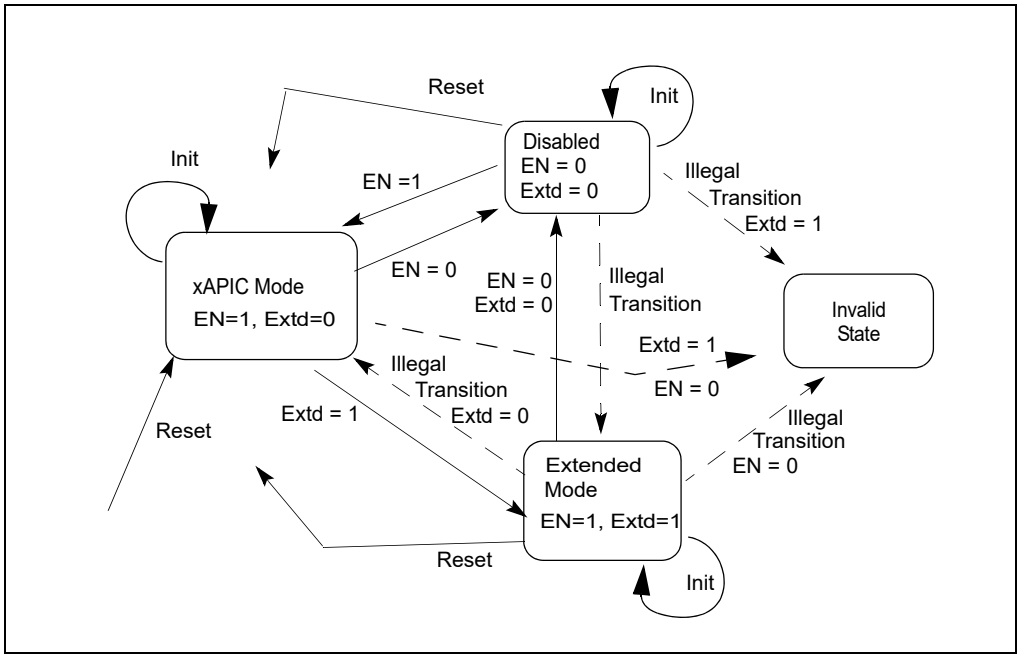


Figure 10-27. Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and Reset

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

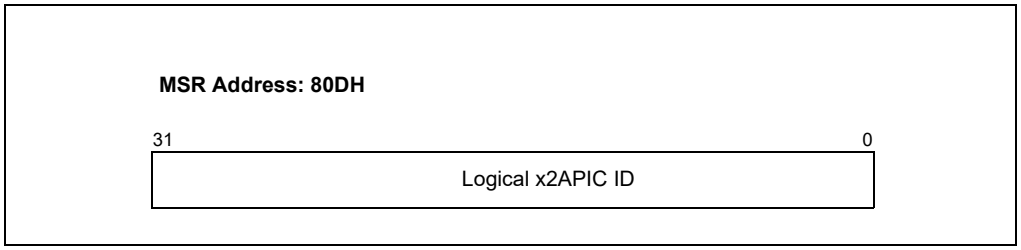


Figure 10-29. Logical Destination Register in x2APIC Mode

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 « x2APIC ID[3:0]. The remaining bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

```
Logical x2APIC ID = [(x2APIC ID[19:4] « 16) | (1 « 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).

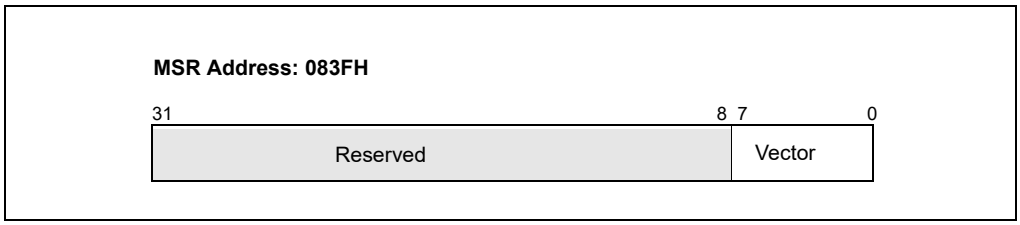


Figure 10-30. SELF IPI register

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

17. Updates to Chapter 17, Volume 3B

Change bars show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Corrected FREEZE_WHILE_SMM name. Minor update to Section 17.4.8.1 "LBR Stack and Intel® 64 Processors ". Correction to Table 17-11 " MSR_LBR_SELECT for Intel microarchitecture code name Nehalem", Table 17-12 "MSR_LBR_SELECT for Intel® microarchitecture code name Sandy Bridge", and Table 17-13 "MSR_LBR_SELECT for Intel® microarchitecture code name Haswell". Various updates throughout the rest of the chapter to add L2 CDP updates.

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 (INT 3) below.
- **Breakpoint-address registers (DR0 through DR3) —** Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6) —** Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7) —** Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS —** Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register —** Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register —** Generates a debug exception (#DB) after every execution of an instruction.
- **Breakpoint instruction (INT 3) —** Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.
- Execution of the breakpoint instruction.

- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

17.2 DEBUG REGISTERS

Eight debug registers (see Figure 17-1 for 32-bit operation and Figure 17-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

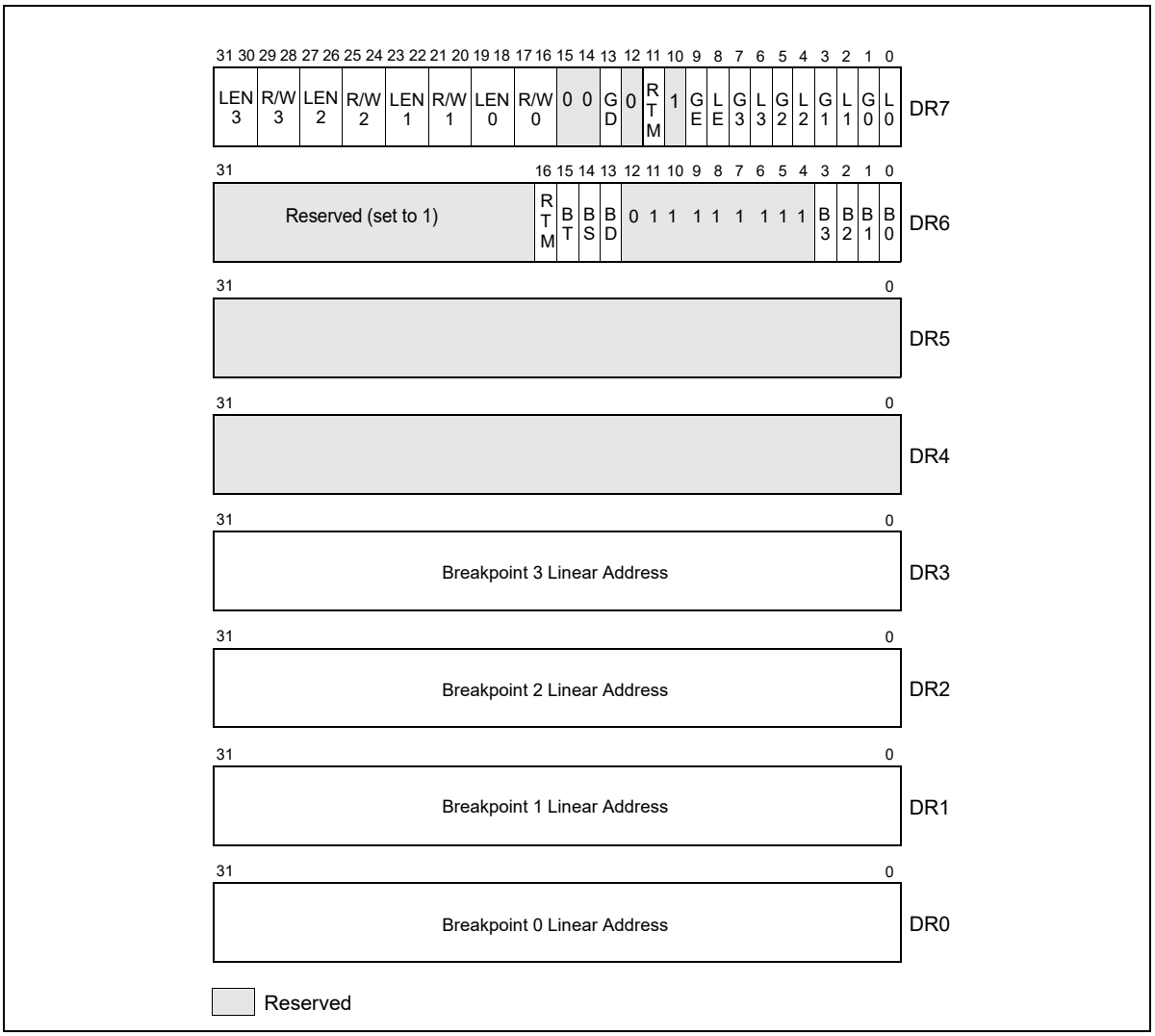


Figure 17-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 though 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 17.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

17.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 17-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

17.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

17.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 17-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3) —** Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LEN*n*, and R/W*n* flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BD (debug register access detected) flag (bit 13) —** Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 17.2.4, "Debug Control Register (DR7)," for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14) —** Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15) —** Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, "Task-State Segment (TSS)," for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when **clear**) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

17.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 17-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6) —** Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated L*n* flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7) —** Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated G*n* flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9) —** This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 17.3.3). This advanced debugging is enabled only if IA32_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13) —** Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

  When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

  The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.
- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29) —** Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/W*n* fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Break on I/O reads or writes.
  11 — Break on data reads or writes but not instruction fetches.

  When the DE flag is clear, the processor interprets the R/W*n* bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Undefined.
  11 — Break on data reads or writes but not instruction fetches.
- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31) —** Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

  00 — 1-byte length.
  01 — 2-byte length.
  10 — Undefined (or 8 byte length, see note below).
  11 — 4-byte length.

If the corresponding RW*n* field in register DR7 is 00 (instruction execution), then the LEN*n* field should also be 00. The effect of using other lengths is undefined. See Section 17.2.5, "Breakpoint Field Recognition," below.

NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LEN*n* field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). The Encoding 10B is supported in processors based on Intel® Atom™ microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

17.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the LEN*n* fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The LEN*n* fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DR*n*). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses LEN*n* field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its LEN*n* field. Table 17-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoint addresses must have a length specification of 1 byte (the LEN*n* field is set to 00). Code breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

Table 17-1. Breakpoint Examples

Debug Register Setup			
Debug Register	**R/W*n***	**Breakpoint Address**	**LEN*n***
DR0 DR1 DR2 DR3	R/W0 = 11 (Read/Write) R/W1 = 01 (Write) R/W2 = 11 (Read/Write) R/W3 = 01 (Write)	A0001H A0002H B0002H C0000H	LEN0 = 00 (1 byte) LEN1 = 00 (1 byte) LEN2 = 01) (2 bytes) LEN3 = 11 (4 bytes)
Data Accesses			
Operation		**Address**	**Access Length (In Bytes)**
Data operations that trap - Read or write - Read or write - Write - Write - Read or write - Read or write - Read or write - Write - Write - Write		 A0001H A0001H A0002H A0002H B0001H B0002H B0002H C0000H C0001H C0003H	 1 2 1 2 4 1 2 4 2 1
Data operations that do not trap - Read or write - Read - Read or write - Read or write - Read - Read or write		 A0000H A0002H A0003H B0000H C0000H C0004H	 1 1 4 2 2 4

17.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 17-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Break point conditions for 8-byte memory read/writes are supported in all modes.

17.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

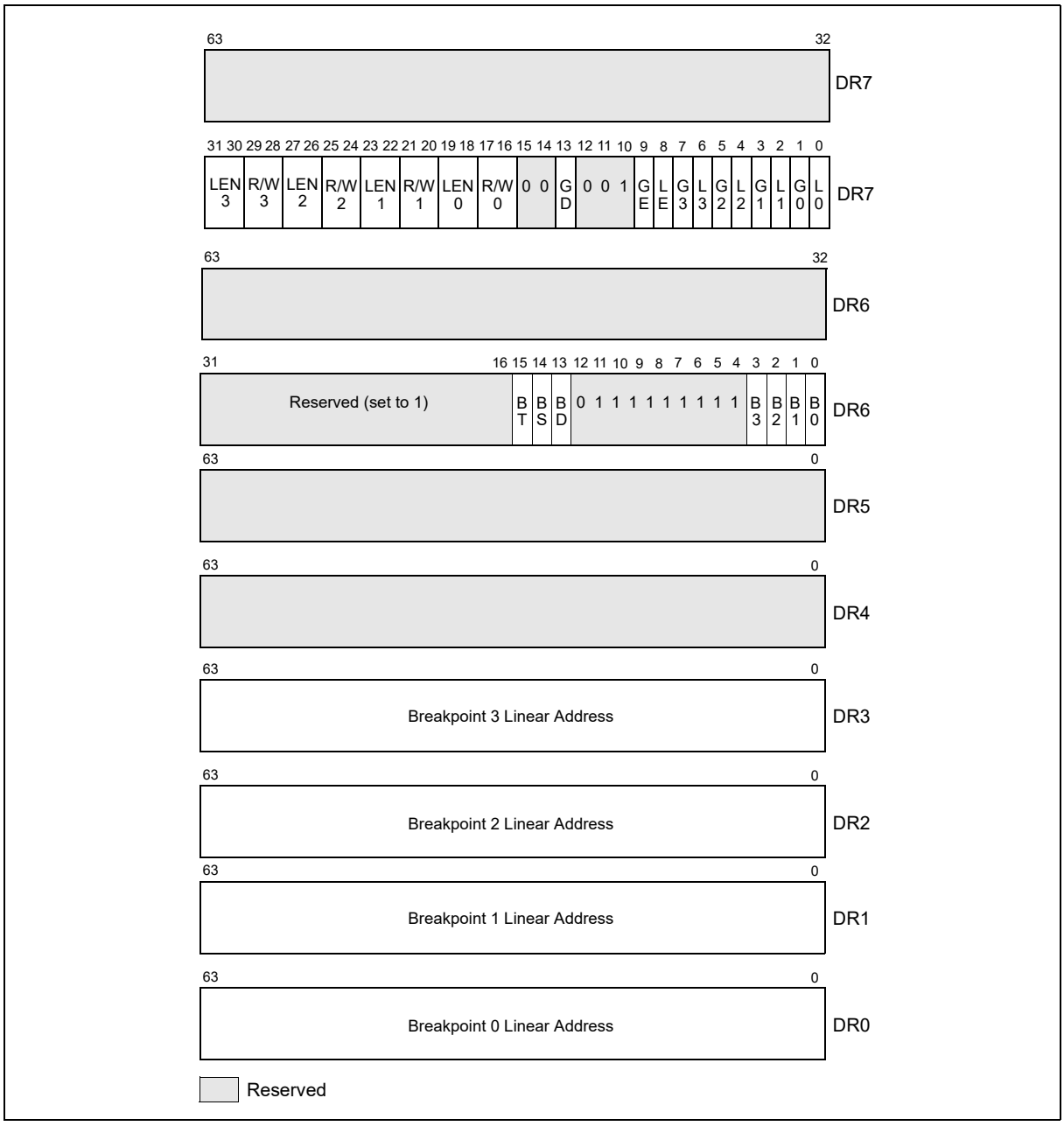


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

17.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 17-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 17.3.1.3, "General-Detect Exception Condition") result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

See also: Chapter 6, "Interrupt 1—Debug Exception (#DB)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Table 17-2. Debug Exception Conditions

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 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/Wn = 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/Wn = 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/Wn = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1		Fault
Task switch	BT = 1		Trap

17.3.1.1 Instruction-Breakpoint Exception Condition

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

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see "MOV—Move" and "POP—Pop a Value from the Stack" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, "System Flags and Fields in the EFLAGS Register," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for code breakpoint, CS limit violation and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - — Debug exceptions generated in response to instruction breakpoints

— Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)

— Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)

— Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

17.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

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

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

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

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

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

17.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

17.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n* and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

17.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

17.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT 3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP)." Debuggers use break exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

17.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify **transactional regions** (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does **not** resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered.

(A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

17.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt and exception recording:

- Section 17.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)"
- Section 17.6, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture"
- Section 17.9, "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.

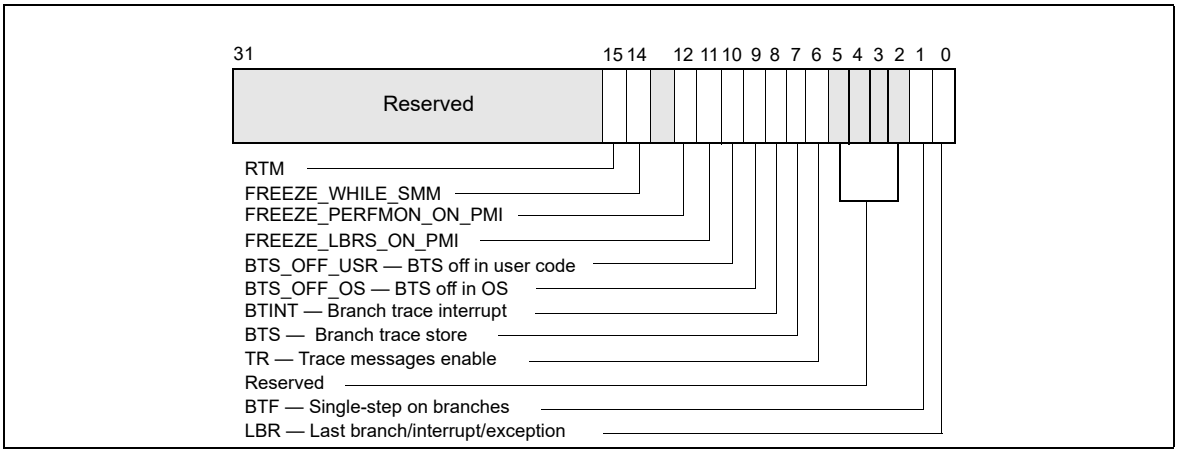


Figure 17-3. IA32_DEBUGCTL MSR for Processors based on Intel Core microarchitecture

- **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).

17.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

17.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 17.4.9.2, "Setting Up the DS Save Area." for additional details.

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, "Task-State Segment (TSS)."

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

17.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if CPUID.01H:ECX[bit 4] = 1.

The CPL-qualified branch trace mechanism is described in Section 17.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, BTS_OFF_USR (bit 10) and BTS_OFF_OS (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

17.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e. CPUID.0AH:EAX[7:0] > 1). These capabilities provides the following interface in IA32_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_LBR_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32_DEBUGCTL. Software must then re-enable IA32_DEBUGCTL.LBR to resume recording branches. When using this feature, software should be careful about writes to IA32_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - Streamlined Freeze_LBR_on_PMI is supported for ArchPerfMonVerID >= 4. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.LBR_Frz =1 to disable recording, but does not change the LBR bit (bit 0) in IA32_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI** (bit 12) — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_Perfmon_On_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32_PERF_GLOBAL_CTRL MSR (see Figure 18-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 18.6.2.1, "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	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.

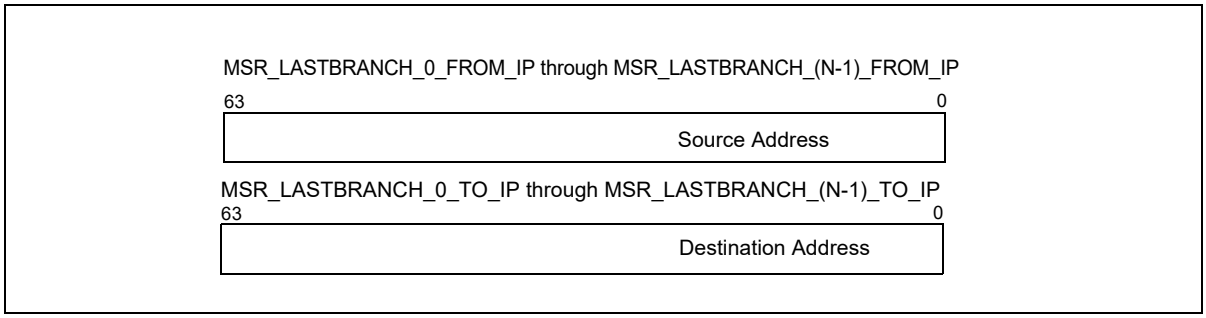


Figure 17-4. 64-bit Address Layout of LBR MSR

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.

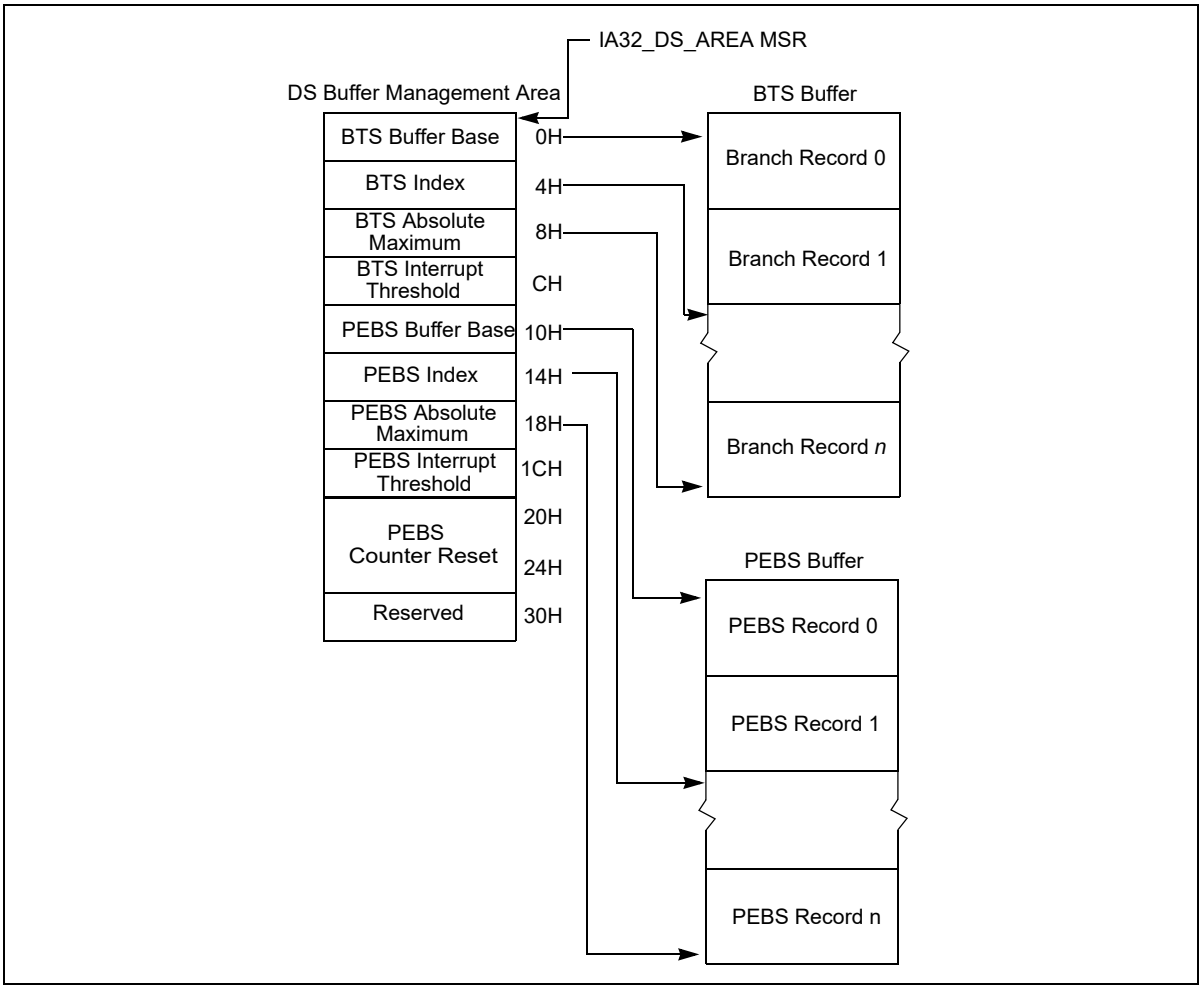


Figure 17-5. DS Save Area Example[1]

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).

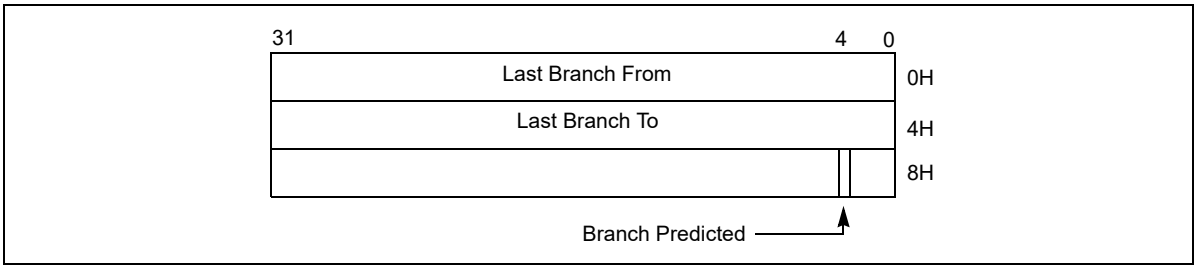


Figure 17-6. 32-bit Branch Trace Record Format

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.

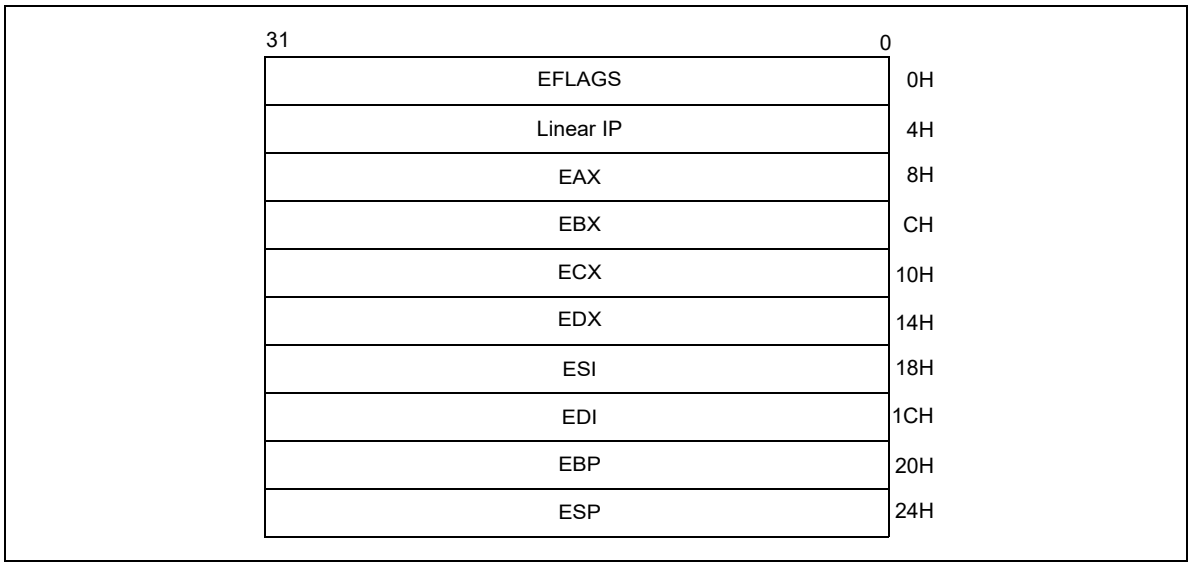


Figure 17-7. PEBS Record Format

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.

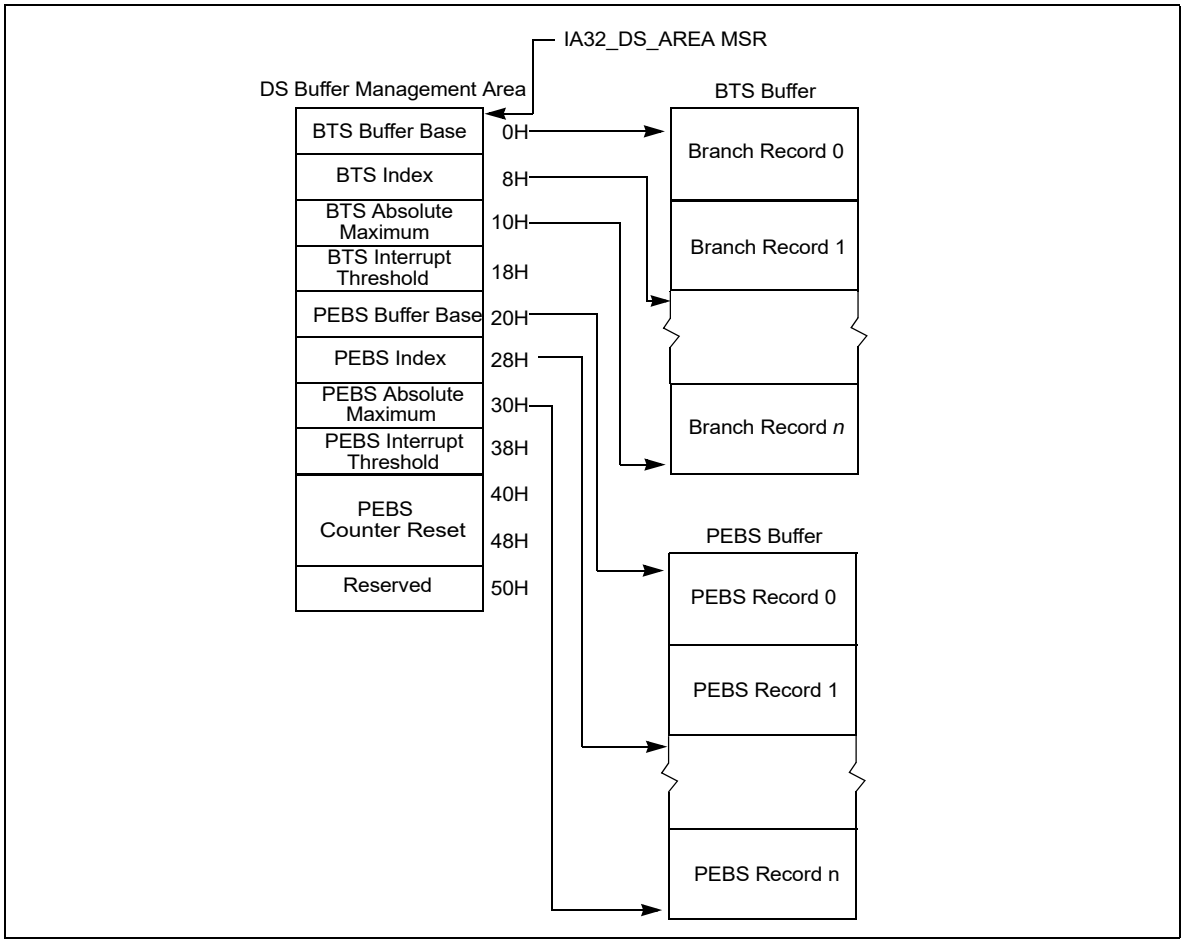


Figure 17-8. IA-32e Mode DS Save Area Example[1]

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).

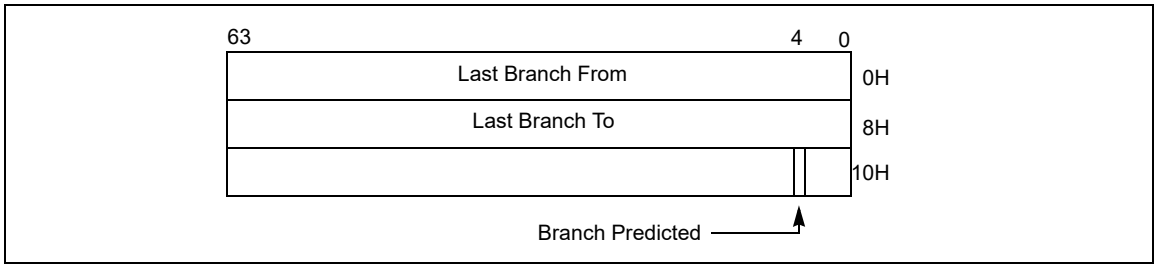


Figure 17-9. 64-bit Branch Trace Record Format

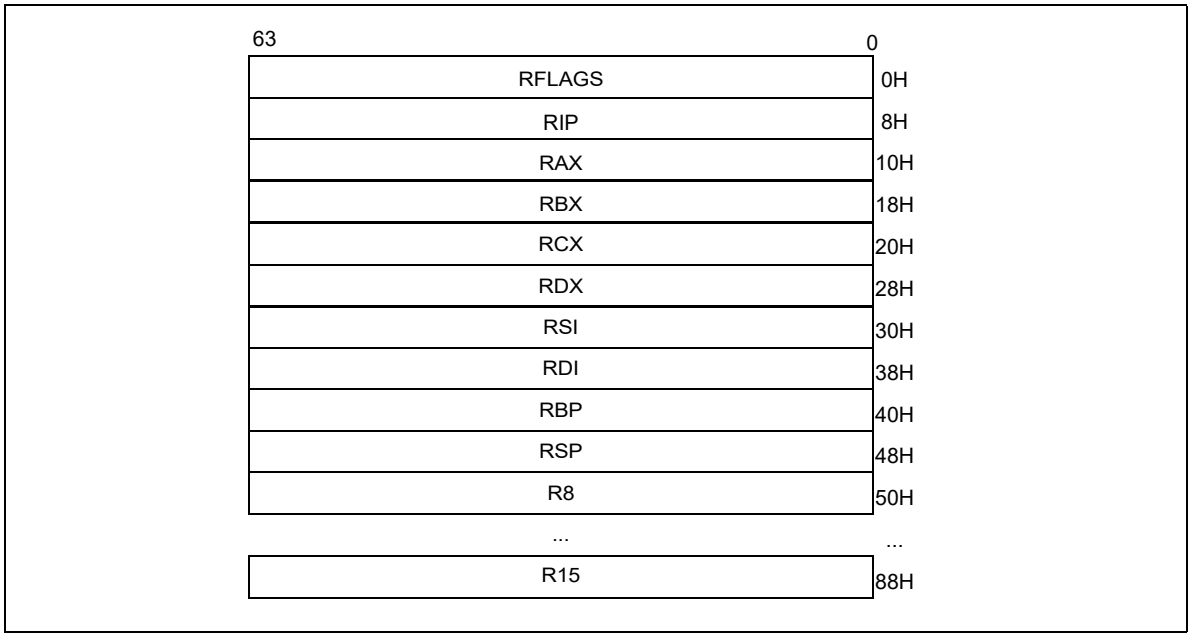


Figure 17-10. 64-bit PEBS Record Format

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 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. MISPRED 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.

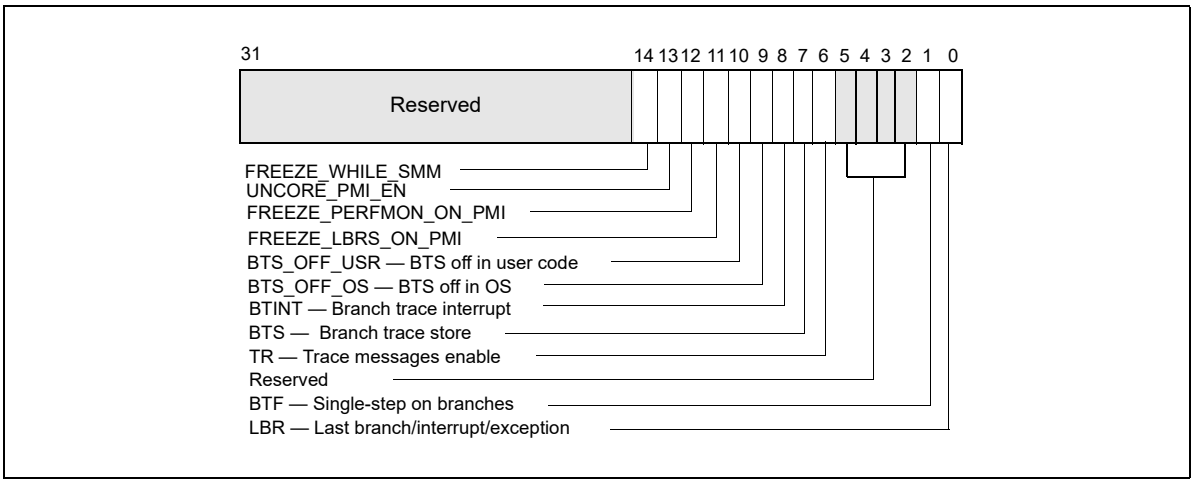


Figure 17-11. IA32_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem

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."

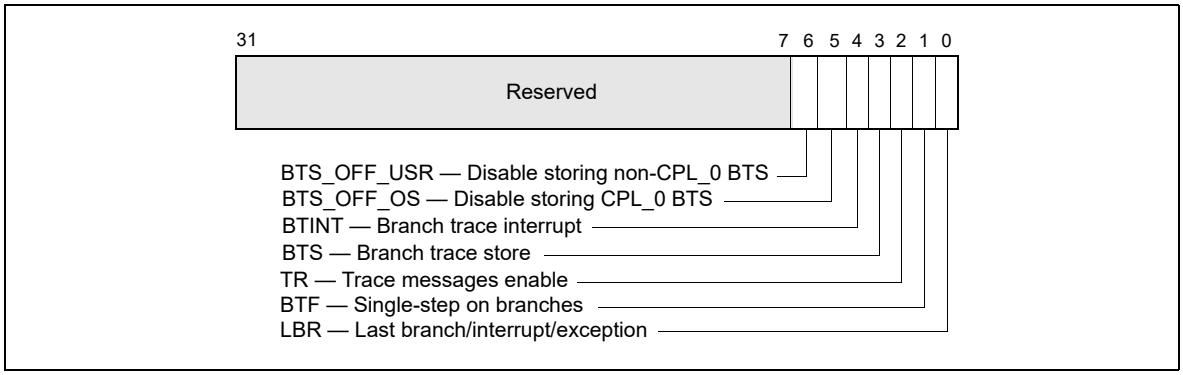


Figure 17-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **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 it maximum value, it wraps around to 0. See Table 17-17 and Figure 17-12.

Table 17-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR_LASTBRANCH_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 17-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the "from" and "to" instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the "from" address is the address of the branch instruction and the "to" address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the "from" address the return instruction pointer (RIP) saved for the interrupt and the "to" address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the "from" address is the linear address of the instruction that caused the exception to be generated and the "to" address is the address of the first instruction in the exception handler routine.

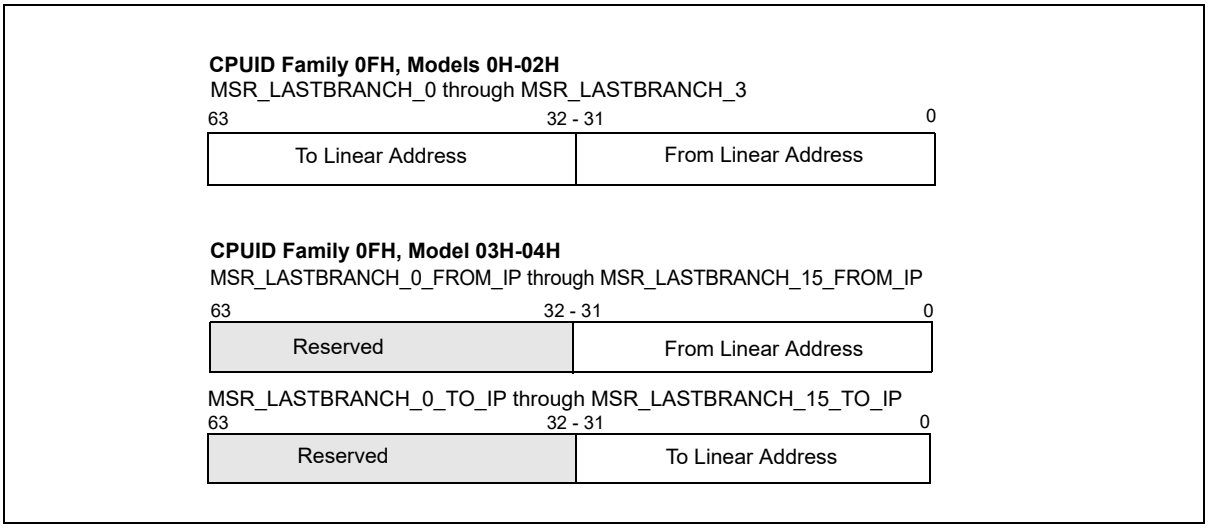


Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

17.13.3 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.

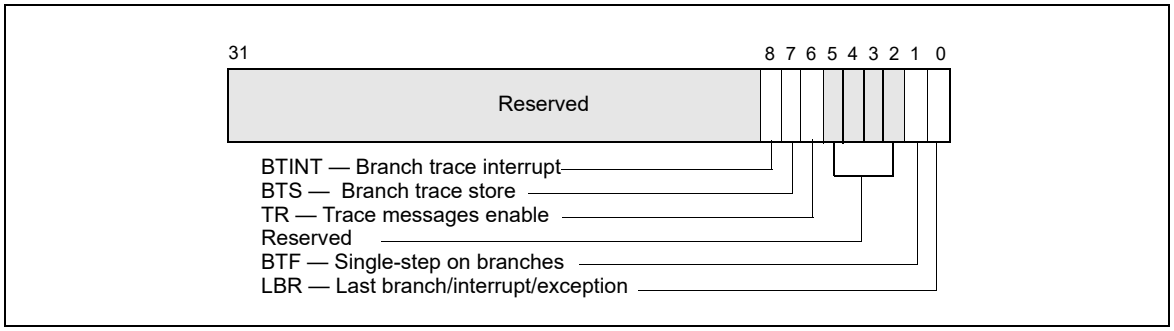


Figure 17-14. IA32_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors

- **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 2.19, "MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

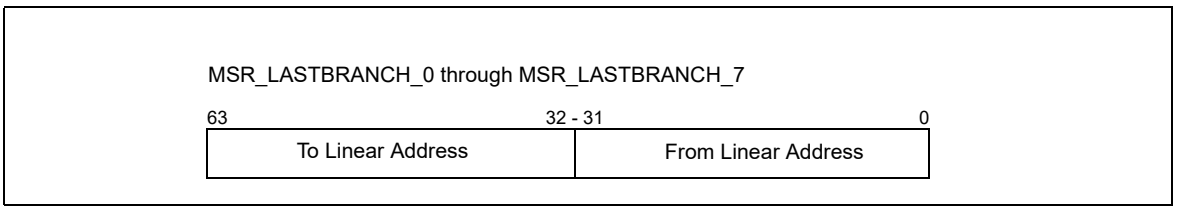


Figure 17-15. LBR Branch Record Layout for the Intel Core Solo and Intel Core Duo Processor

17.15 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.
  - **PB*i* (performance monitoring / breakpoint pins) flags (bits 5-2) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."
  - **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

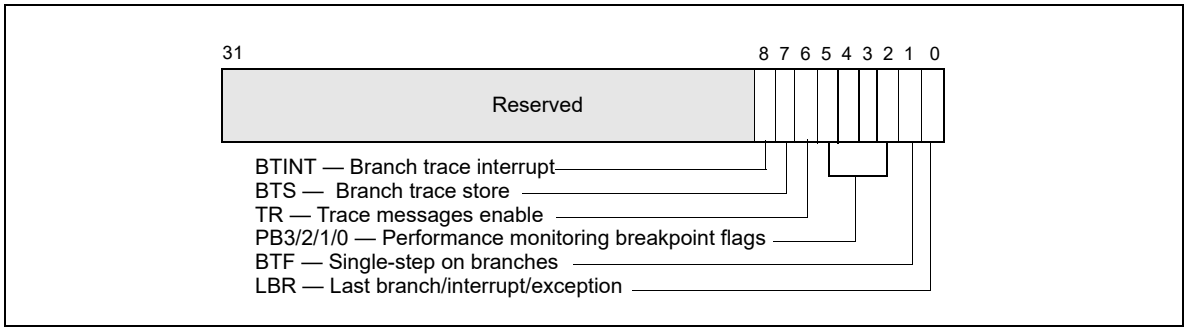


Figure 17-16. MSR_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 17-17.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

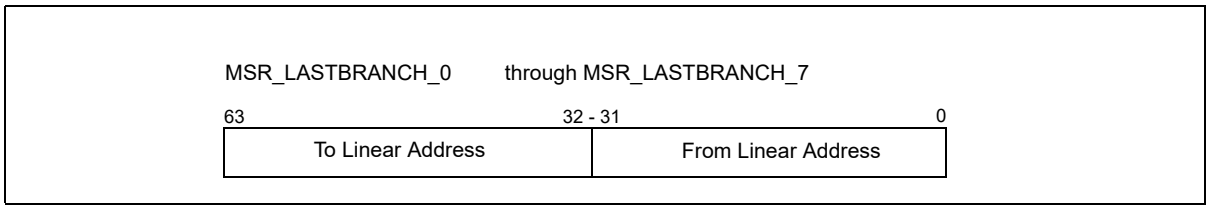


Figure 17-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 17.13.3, "Last Exception Records," and Section 2.20, "MSRs In the Pentium M Processor" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

17.16 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.

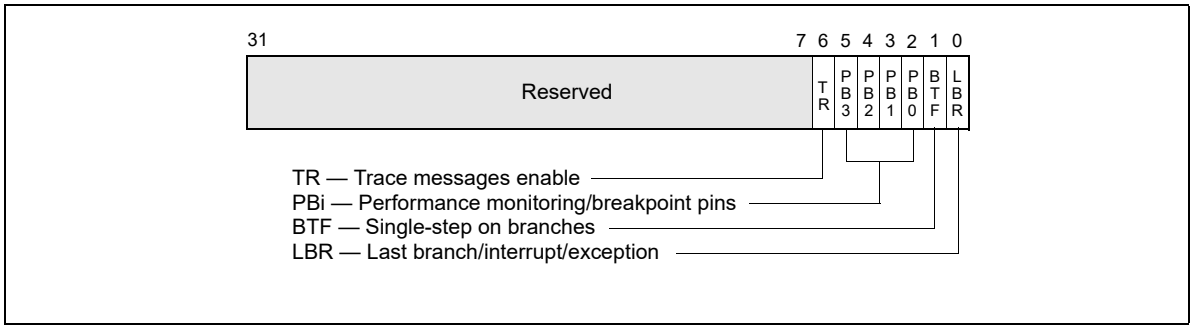


Figure 17-18. DEBUGCTLMSR Register (P6 Family Processors)

- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag. See Section 17.4.3, "Single-Stepping on Branches."
- **PB*i* (performance monitoring / breakpoint pins) flags (bits 2 through 5) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
- **TR (trace message enable) flag (bit 6) —** When set, trace messages are enabled as described in Section 17.4.4, "Branch Trace Messages." Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

17.16.2 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 DEBUGCTLMSR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the LastBranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

17.17 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 than ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32_TSC_ADJUST MSR (address 3BH). Like the IA32_TIME_STAMP_COUNTER MSR, the IA32_TSC_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32_TSC_ADJUST MSR as follows:

- On RESET, the value of the IA32_TSC_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32_TIME_STAMP_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32_TSC_ADJUST MSR.
- If an execution of WRMSR to the IA32_TSC_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32_TSC_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32_TIME_STAMP_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32_TSC_ADJUST MSR on each logical processor.

Processor support for the IA32_TSC_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC_ADJUST (bit 1).

17.17.4 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:

TSC_Value = (ART_Value * CPUID.15H:EBX[31:0])/ CPUID.15H:EAX[31:0] + K

Where 'K' is an offset that can be adjusted by a privileged agent[2].

When ART hardware is reset, both invariant TSC and K are also reset.

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 ID**s (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.

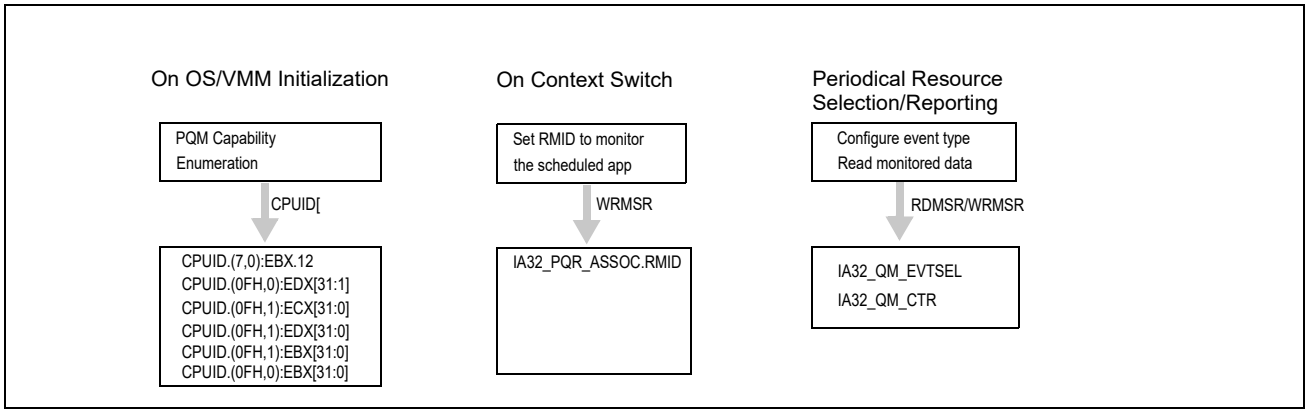


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

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.

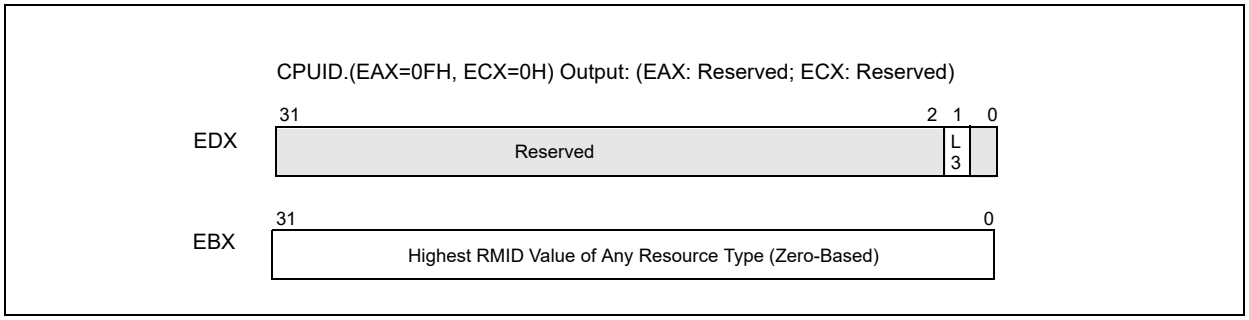


Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

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.

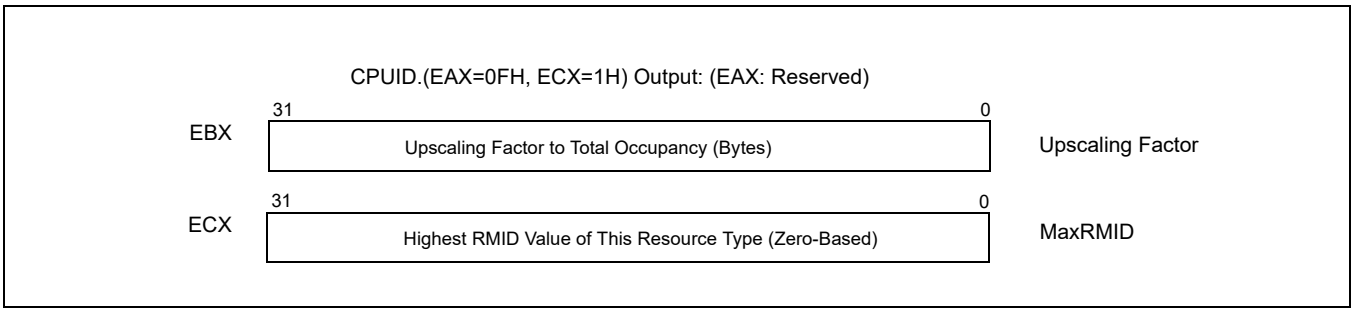


Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))

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.

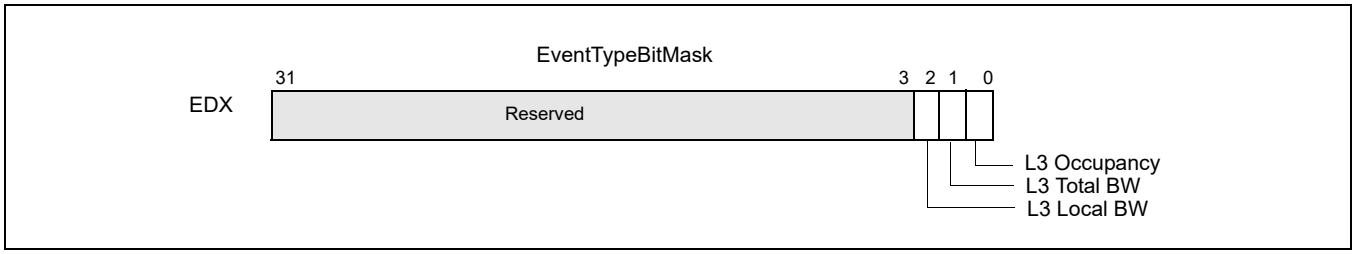


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))

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 + CPUID.(EAX=0FH, ECX=0):EBX[31:0])). The value of IA32_PQR_ASSOC after power-on is 0.

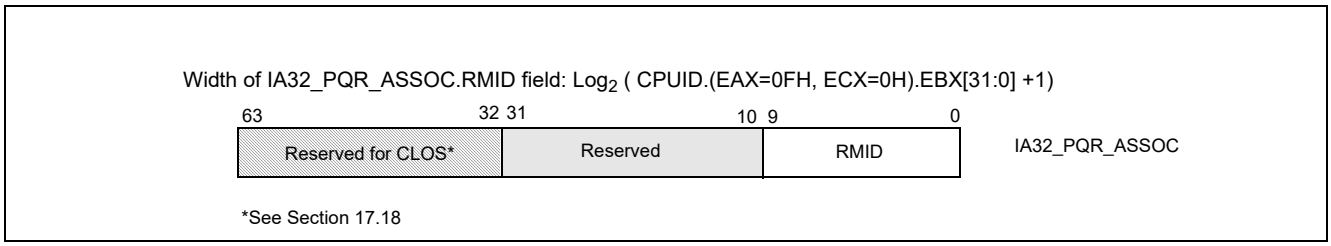


Figure 17-23. IA32_PQR_ASSOC MSR

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32_PQR_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g. RMID 31) may only be supported by one resource type but not by another resource type.

17.18.7 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.

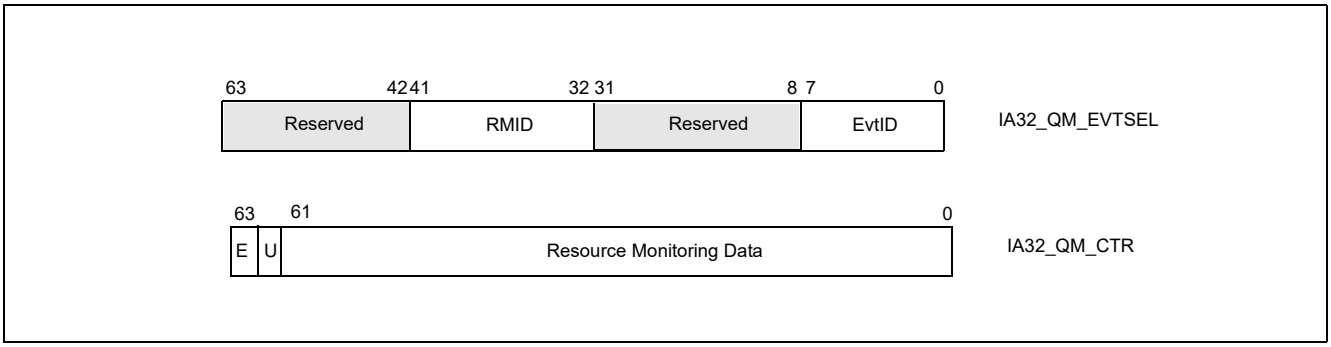


Figure 17-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs

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.

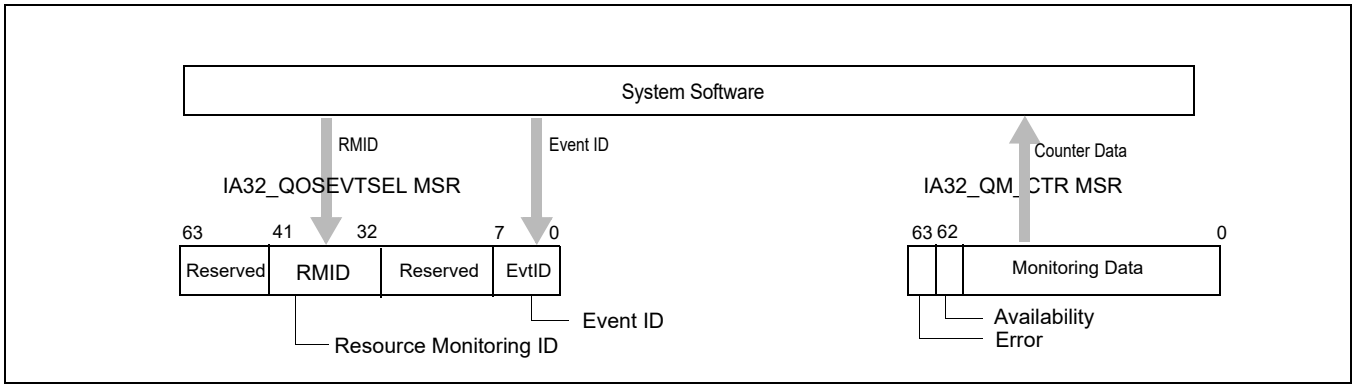


Figure 17-25. Software Usage of Cache Monitoring Resources

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.

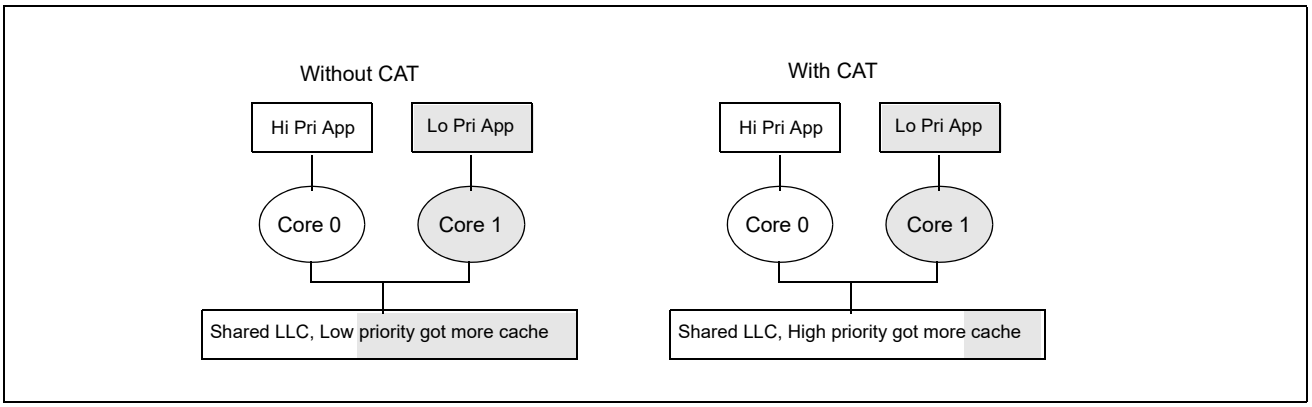


Figure 17-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications

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.

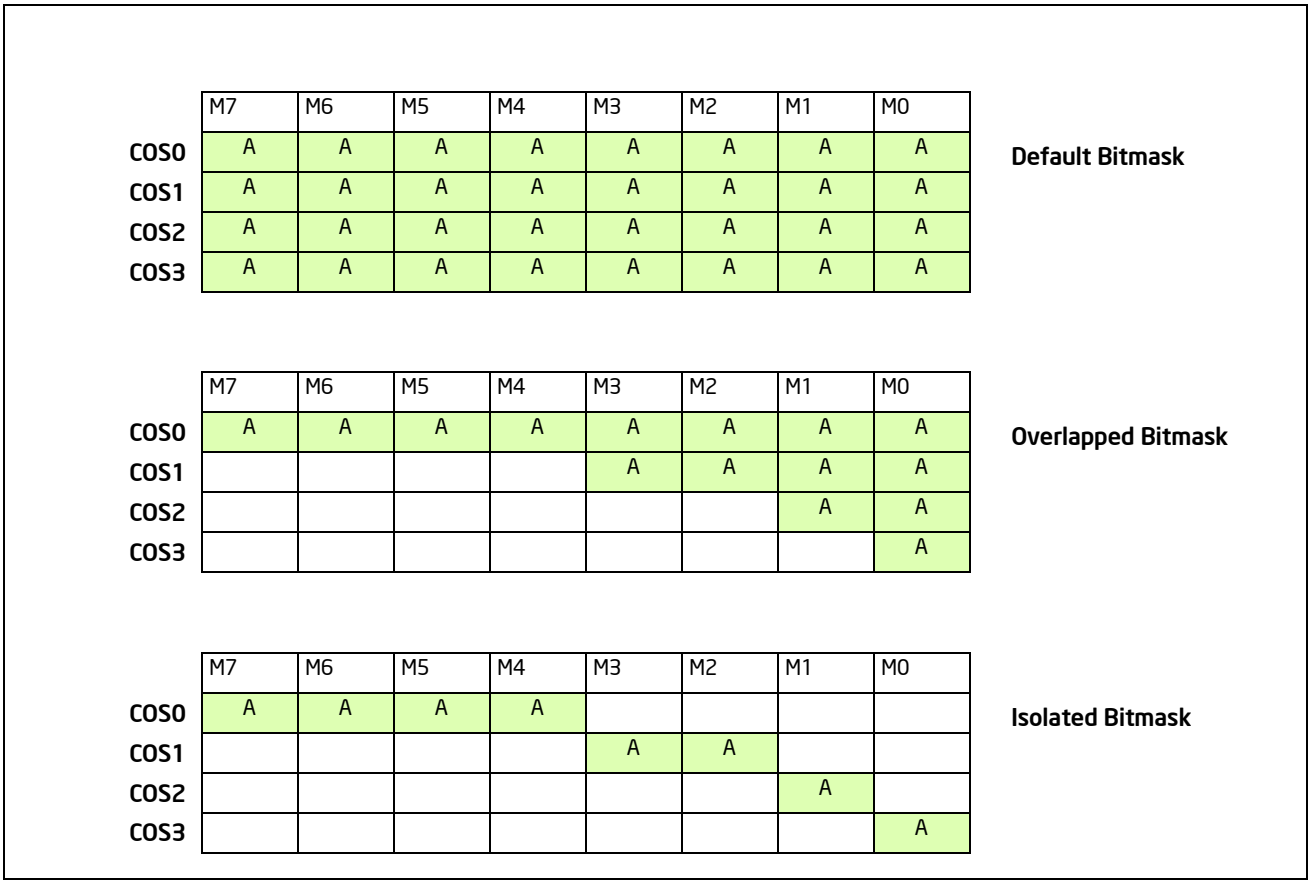


Figure 17-27. Examples of Cache Capacity Bitmasks

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.

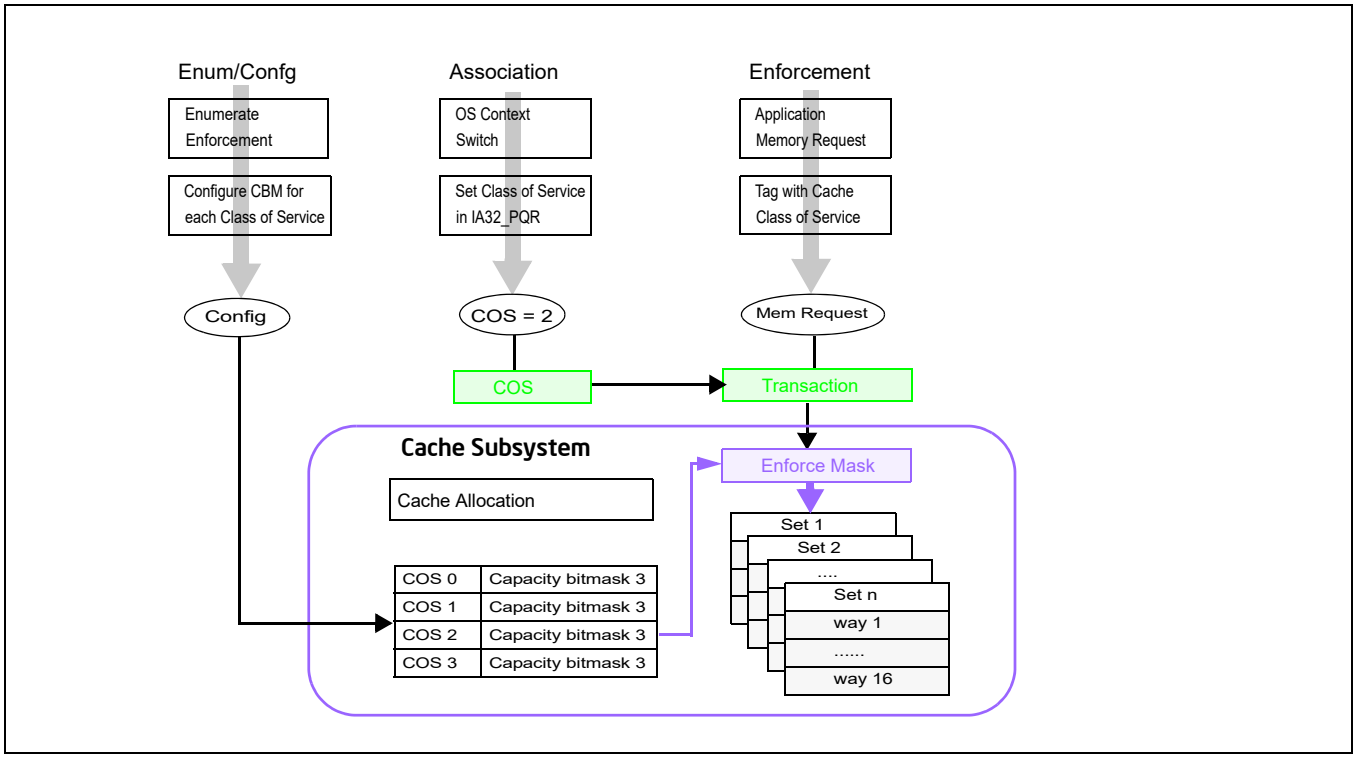


Figure 17-28. Class of Service and Cache Capacity Bitmasks

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.

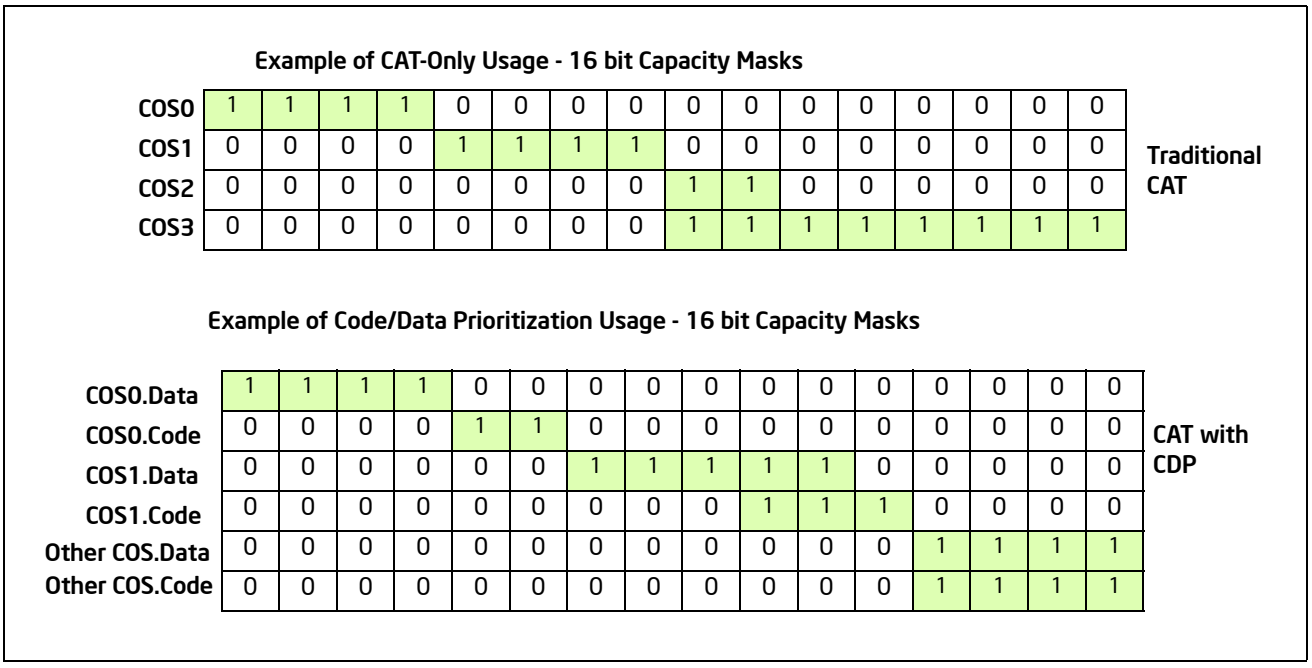


Figure 17-29. Code and Data Capacity Bitmasks of CDP

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.

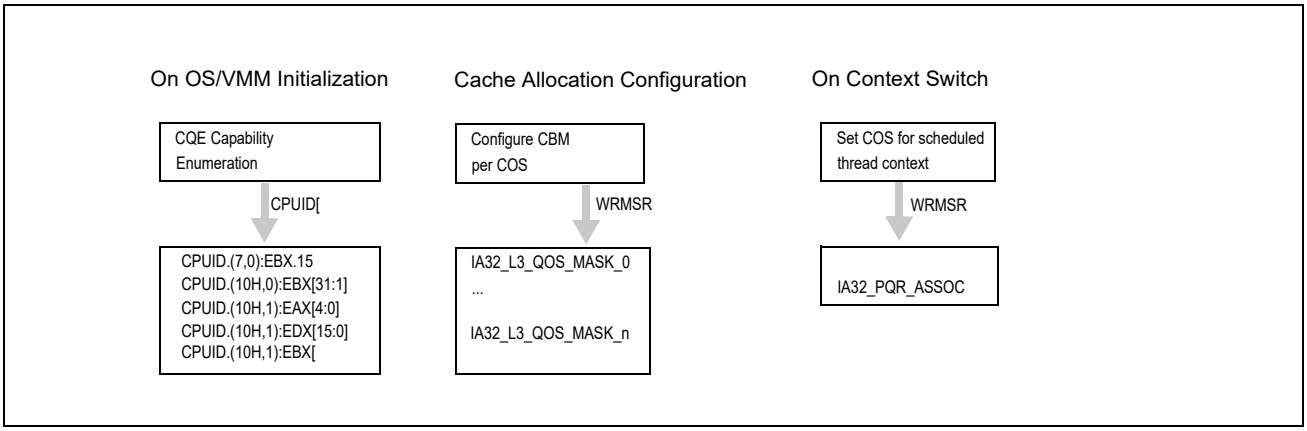


Figure 17-30. Cache Allocation Technology Usage Flow

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 CMCI and Machine Check Resources”. Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.
- IA32_PQR_ASSOC.CLOS: The IA32_PQR_ASSOC MSR provides a COS field that OS/VMM can use to assign a logical processor to an available COS. The set of COS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software COS management overhead at context swap time. See Section 17.19.4.4 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).

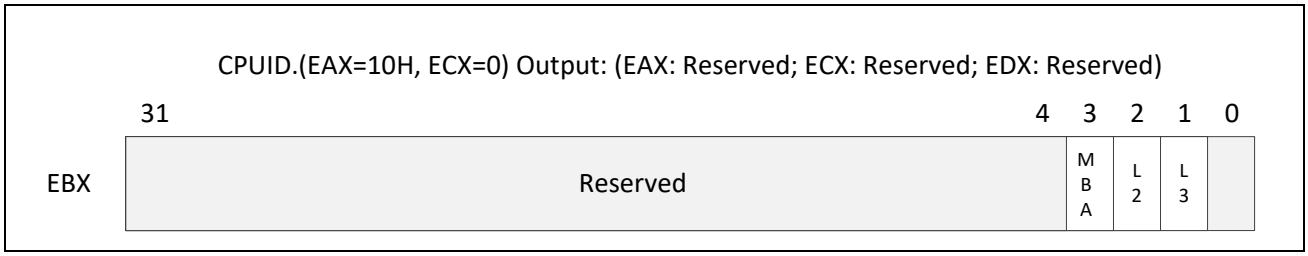


Figure 17-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

— 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:

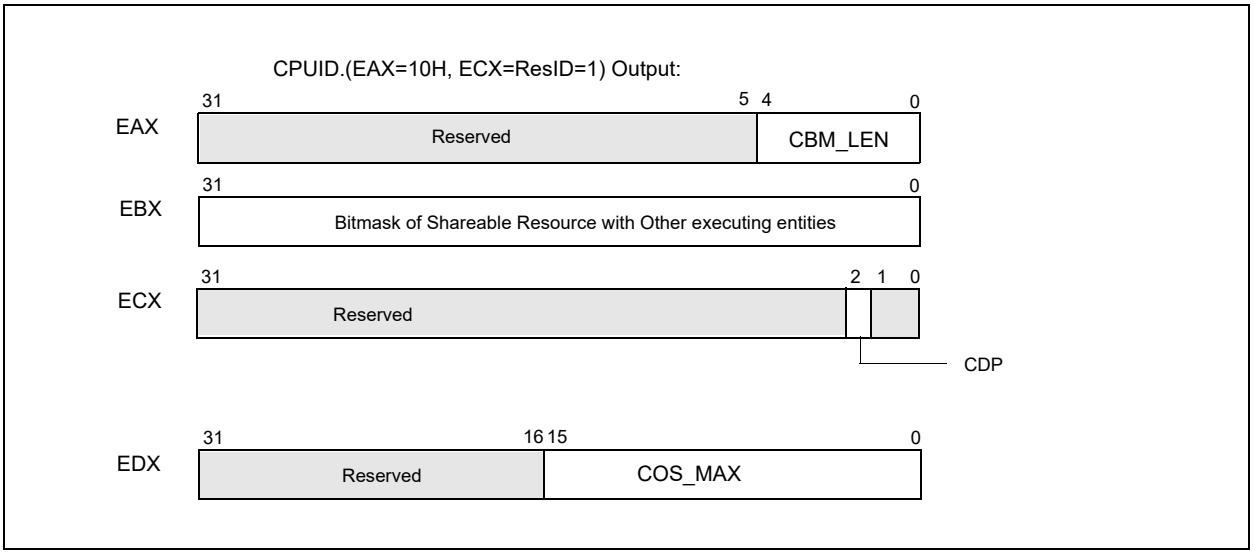


Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration

— 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:

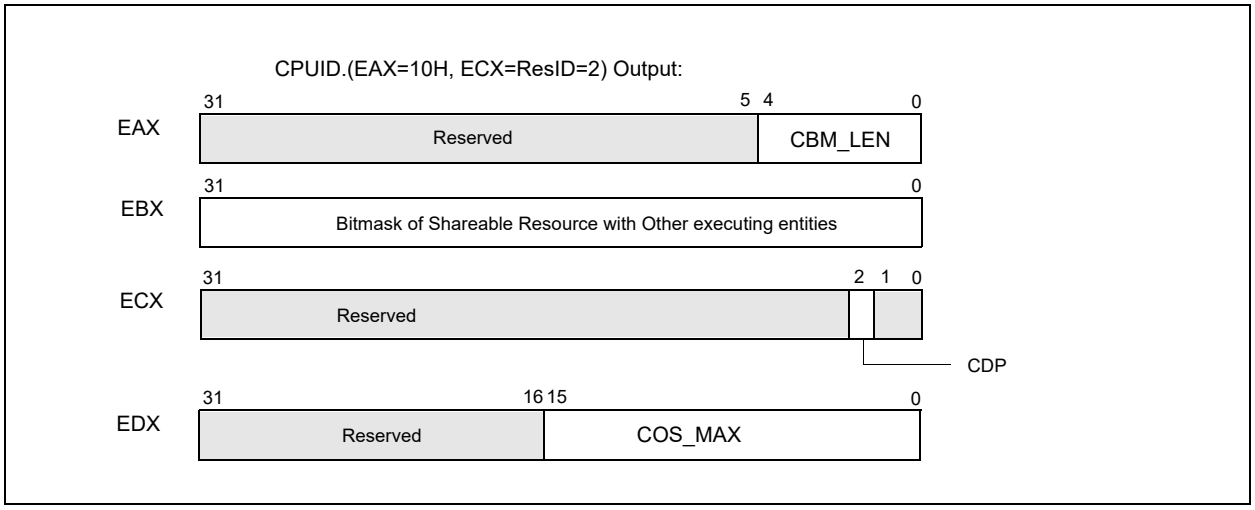


Figure 17-33. L2 Cache Allocation Technology

— 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 MSRs is reserved for Cache Allocation Technology registers of the form IA32_resourceType_MASK_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resourceType is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

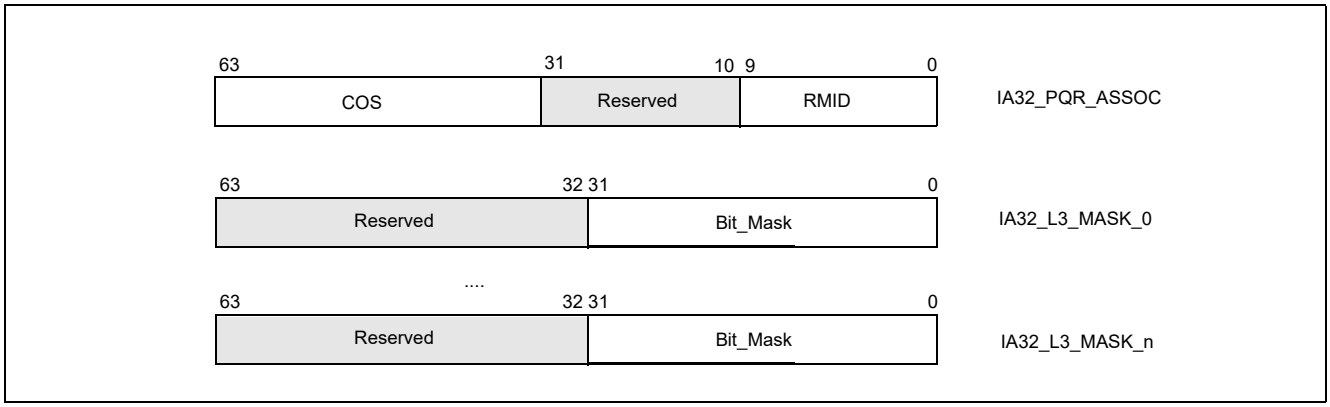


Figure 17-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32_L2_MASK_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 17-34 and Figure 17-35 provide an overview of the relevant registers.

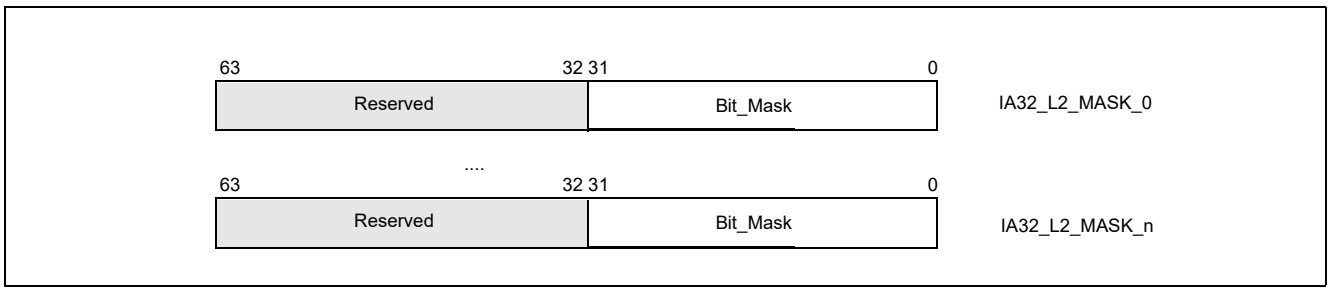


Figure 17-35. IA32_L2_MASK_n MSRs

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (COS) using the IA32_PQR_ASSOC MSR as described in Chapter 17, “Class of Service to Cache Mask Association: Common Across Allocation Features”.

17.19.4.4 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).

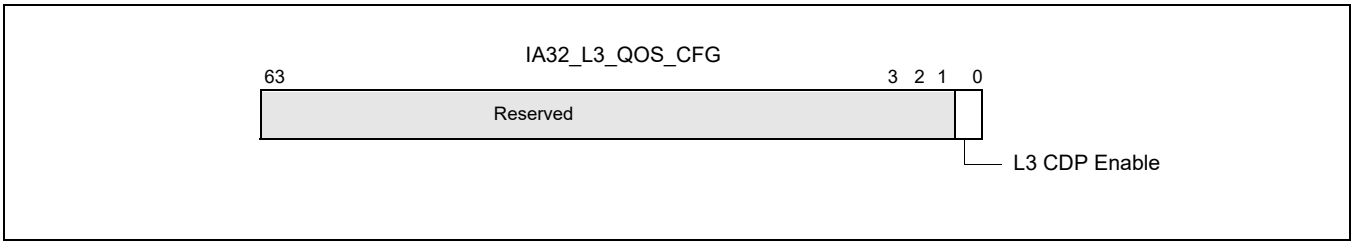


Figure 17-36. Layout of IA32_L3_QOS_CFG

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).

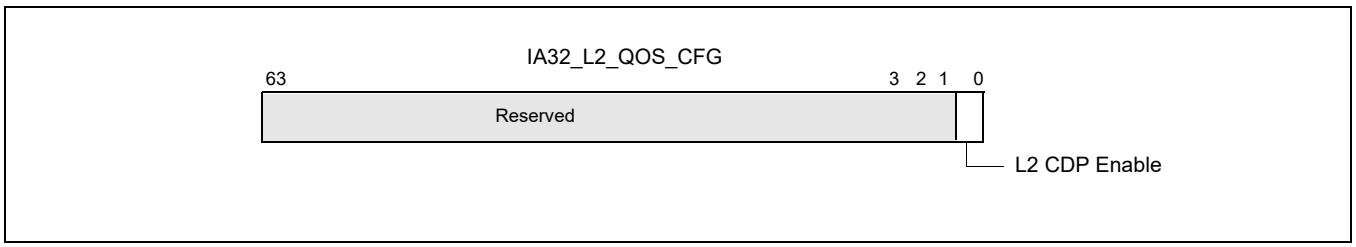


Figure 17-37. Layout of IA32_L2_QOS_CFG

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.

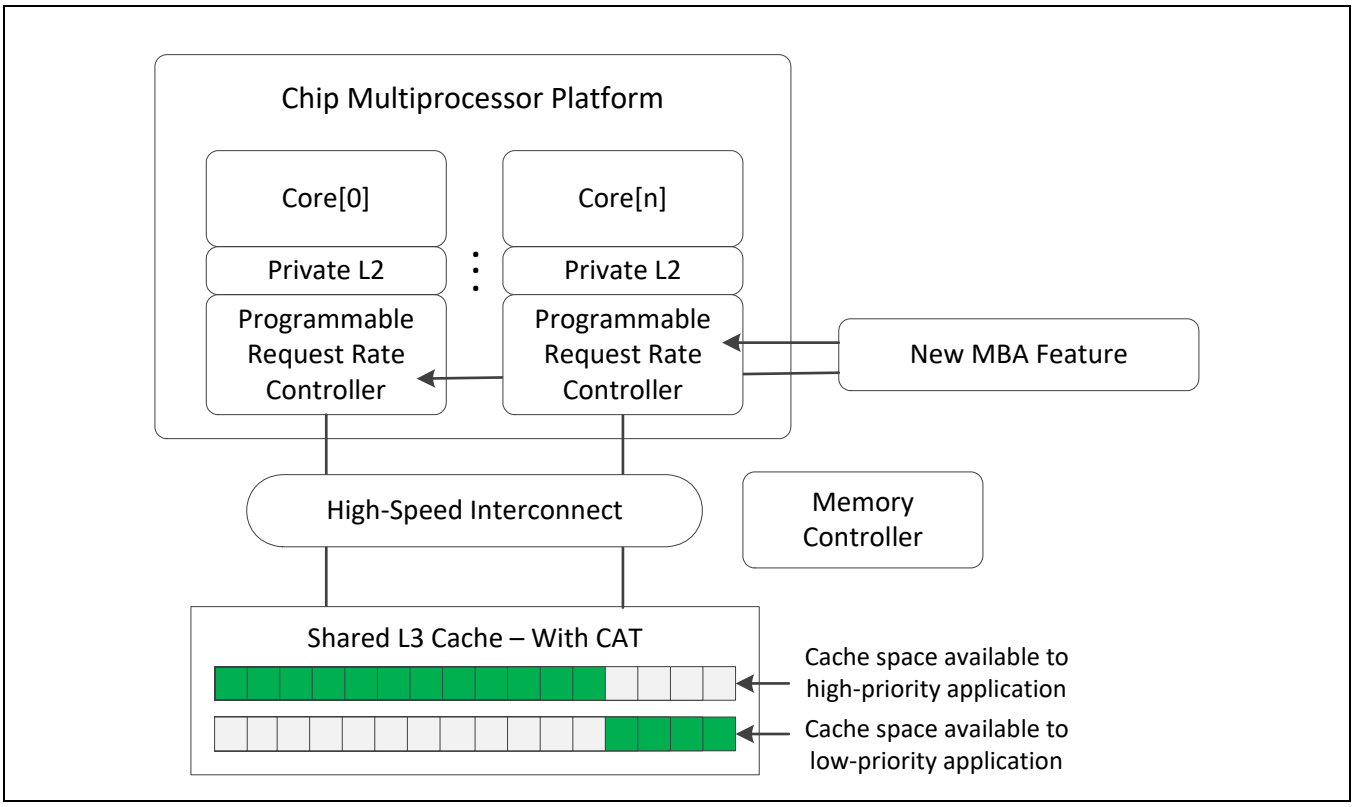


Figure 17-38. A High-Level Overview of the MBA Feature

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.

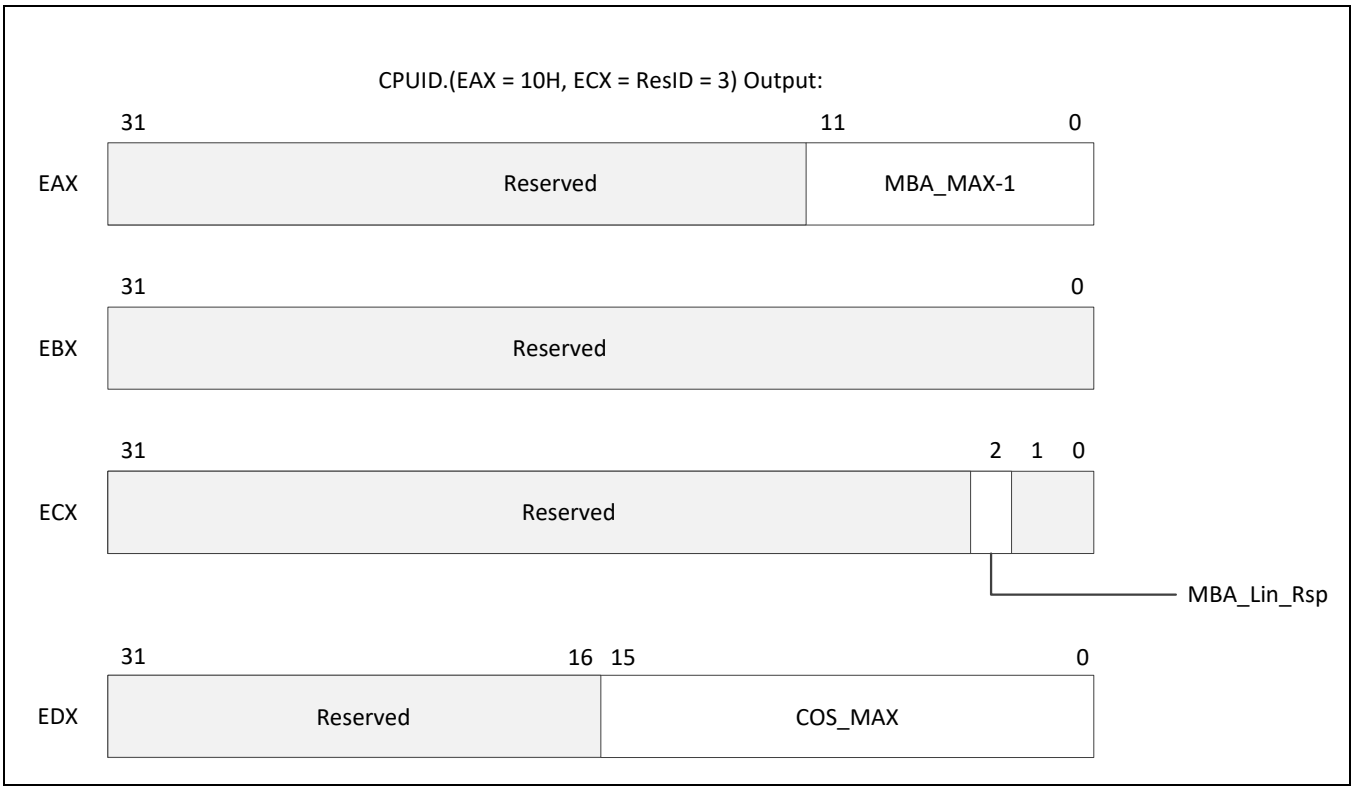


Figure 17-39. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification

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.

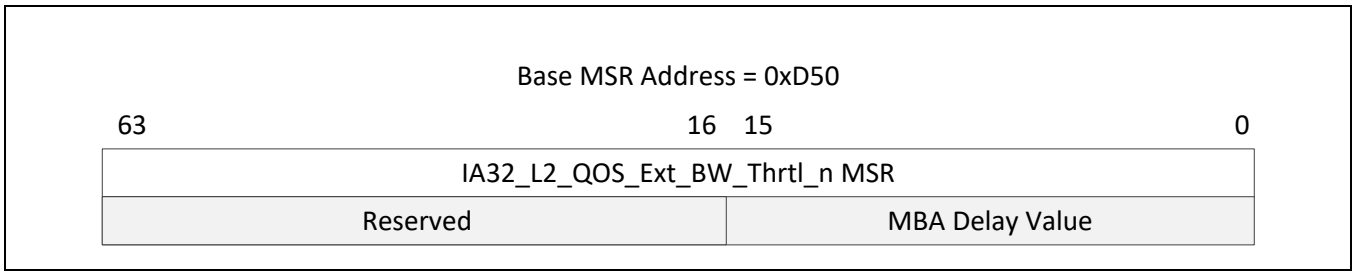


Figure 17-40. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition

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 inadver-tent throttling of the high-priority thread.

18. 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.

Change to chapter: Updated FREEZE_WHILE_SMM name.

CHAPTER 18
PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

18.1 PERFORMANCE MONITORING OVERVIEW

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

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

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

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.6.3, "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"

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 micro-architecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microar-chitectures support version ID 1, 2, and 3. Intel processors based on the Skylake and Kaby Lake microarchitectures support versionID 4.

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support versionID 4.

18.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- Bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR)
- Number of bits supported in each IA32_PMCx
- Number of architectural performance monitoring events supported in a logical processor

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/ IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

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

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

See Figure 18-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7) —** Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

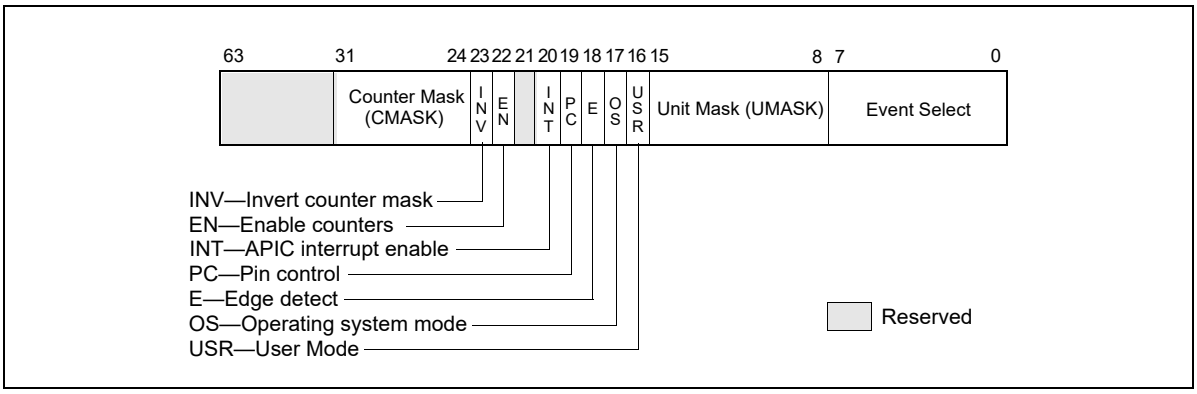


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX. Architectural performance events available in the initial implementation are listed in Table 19-1.
- **USR (user mode) flag (bit 16) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19) —** When set, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.

- **Counter mask (CMASK) field (bits 24 through 31) —** When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

18.2.1.2 Pre-defined Architectural Performance Events

Table 18-1 lists architecturally defined events.

Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

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

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

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

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

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

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

  — HLT

  — STPCLK# pin asserted

  — being throttled by TM1

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

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

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

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

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

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

  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

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

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

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

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

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

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

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

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

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

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

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

NOTE

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

18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.

- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - — IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - — IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - — IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - — IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - — IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

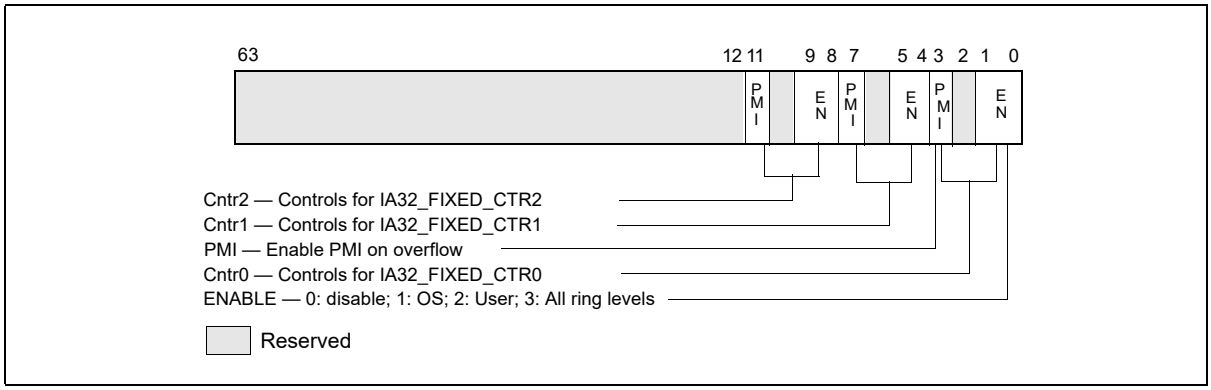


Figure 18-2. Layout of IA32_FIXED_CTR_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

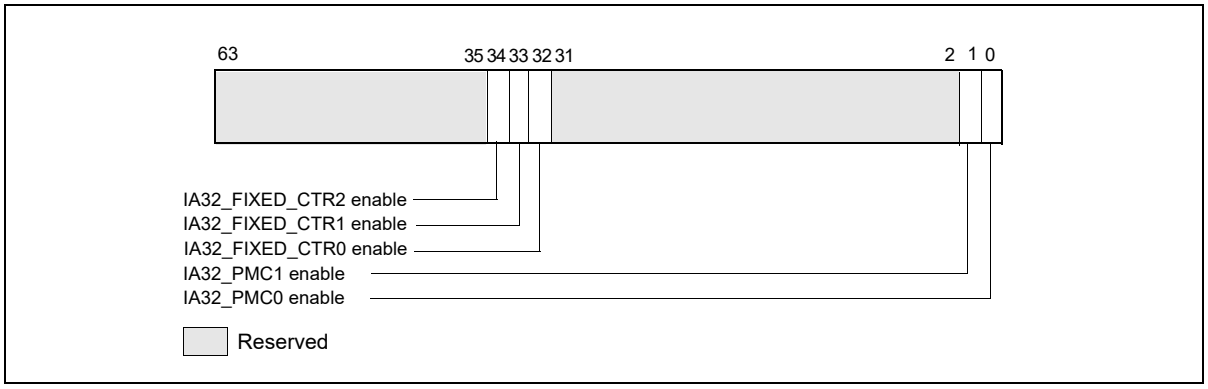


Figure 18-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

Table 18-2. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 18-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status., and sets the "OvfBuffer" bit in IA32_PERF_GLOBAL_STATUS.

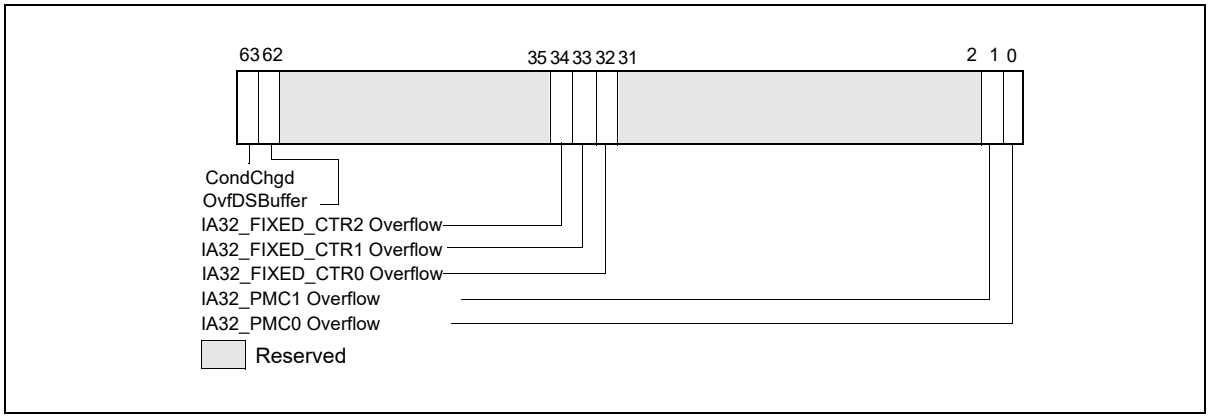


Figure 18-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 18-5.

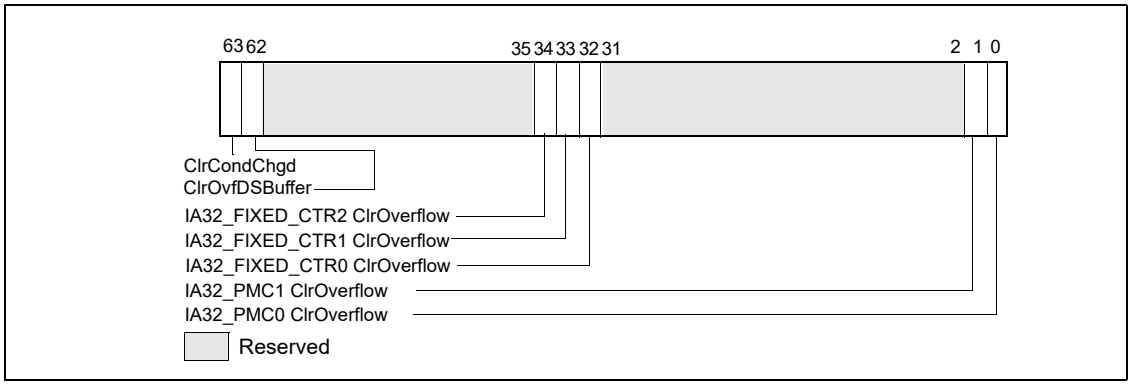


Figure 18-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

18.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 18-6.

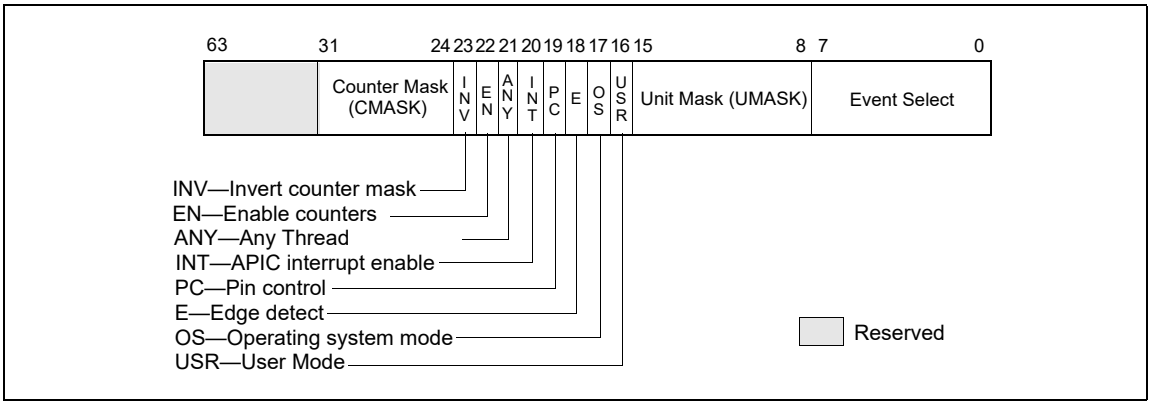


Figure 18-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

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

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

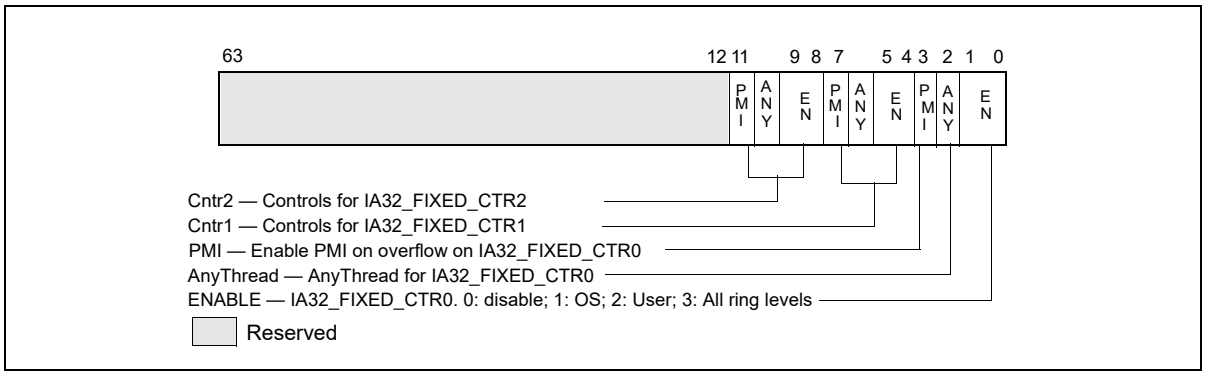


Figure 18-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

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

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 18-8 and Figure 18-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

Note: The number of general-purpose performance monitoring counters (i.e. N in Figure 18-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g. N=2 for 45 nm Intel Atom processors; N =4 for processors based on the Nehalem microarchitecture; for processors based on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active).

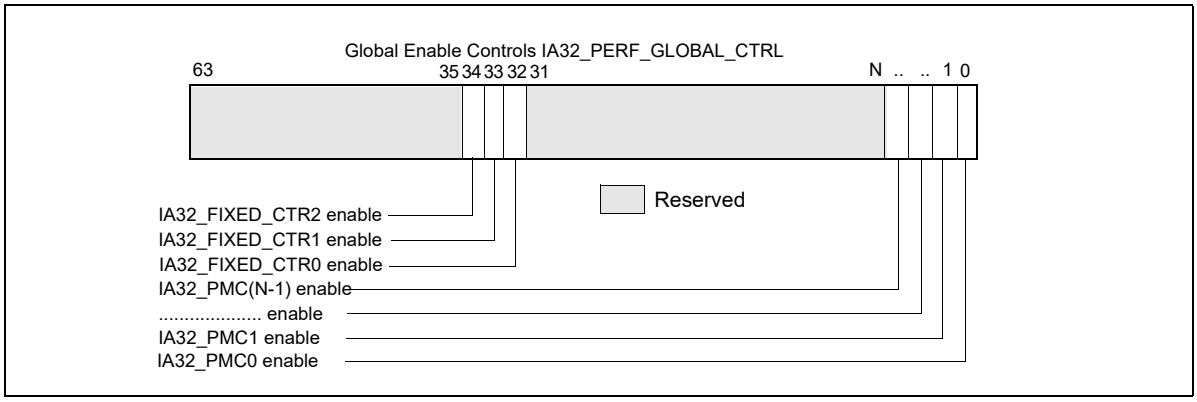


Figure 18-8. Layout of Global Performance Monitoring Control MSR

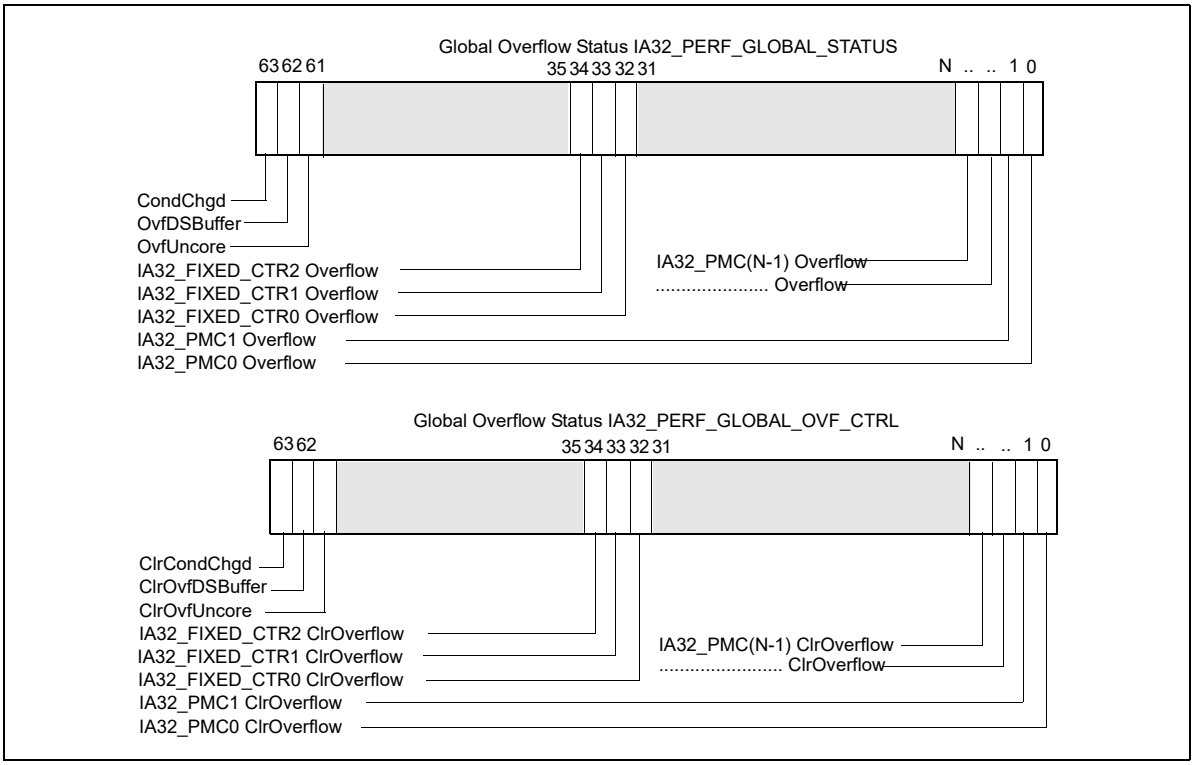


Figure 18-9. Global Performance Monitoring Overflow Status and Control MSRs

18.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 23, "Introduction to Virtual-Machine Extensions") where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see CHAPTER 24 for additional information),
- clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see CHAPTER 24, CHAPTER 26, and CHAPTER 27).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted (see relevant sections of Chapter 19, "Performance Monitoring Events").

18.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancement:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 18.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 18.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 18.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

18.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  - — IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  - — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serve as an read-only control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:
  - — (IA32_DEBUGCTL.LBR & (!IA32_PERF_GLOBAL_STATUS.LBR_FRZ)) =1

- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:
  - — IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and
  - — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:
    - (IA32_PERFEVTSELn.EN & IA32_PERF_GLOBAL_CTRL.PMCn & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for programmable counter ‘n’, or
    - (IA32_PERF_FIXED_CRTL.ENi & IA32_PERF_GLOBAL_CTRL.FCi & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for fixed counter ‘i’

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeza_Perfmon_On_PMI to filter out data that may distort target workload analysis, see Table 17-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g. race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support SGX (For additional information about Intel SGX, see “Intel® Software Guard Extensions Programming Reference”.):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e. IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

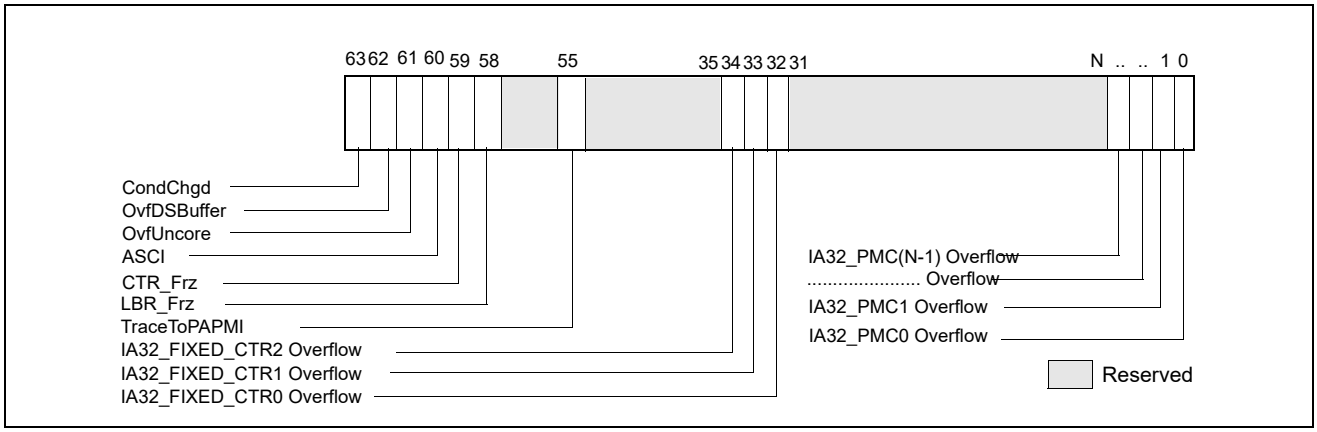


Figure 18-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural Perfmon Version 4

Note, a processor’s support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

18.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_GLOBAL_STATUS_RESET (see Figure 18-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 18.2.4.1.

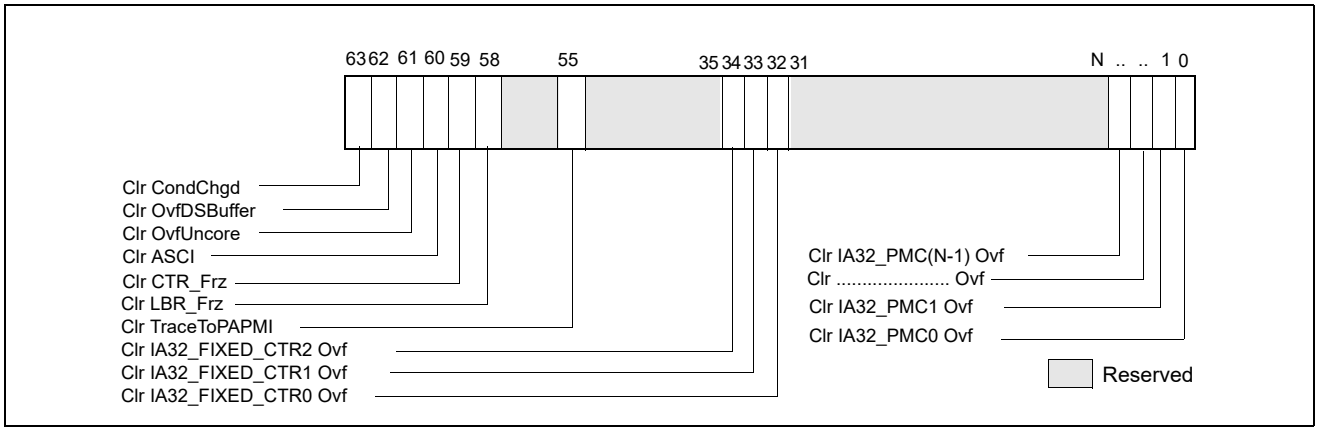


Figure 18-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

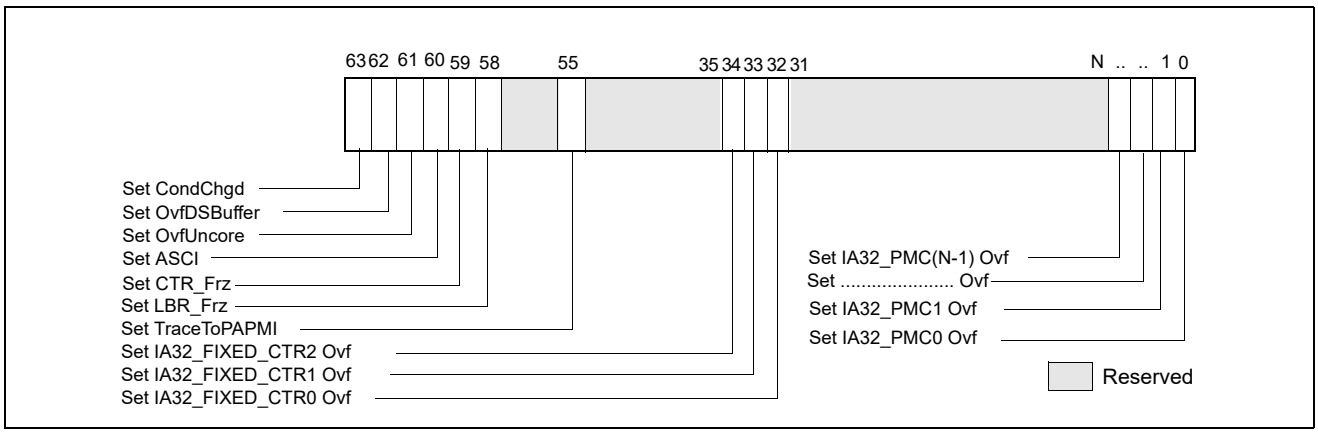


Figure 18-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4

18.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor's performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve

the need of multiple agent adequately. A white paper, "Performance Monitoring Unit Sharing Guideline"[1], proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 18-13.

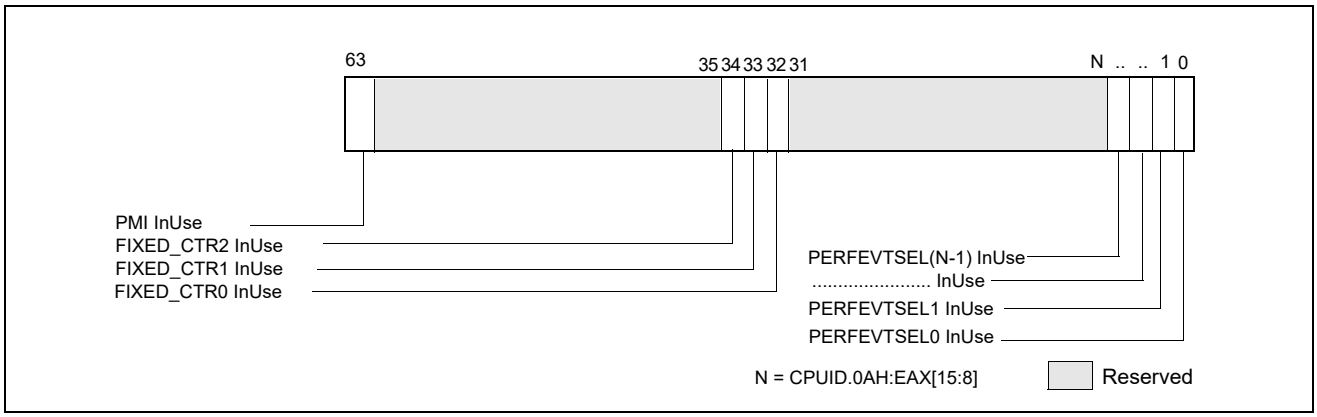


Figure 18-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural Perfmon Version 4

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.

1. Available at http://www.intel.com/sdm

18.2.5 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 18-63.

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 family[2] supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-29. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

2. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem; the performance monitoring facilities described in this section generally also apply.

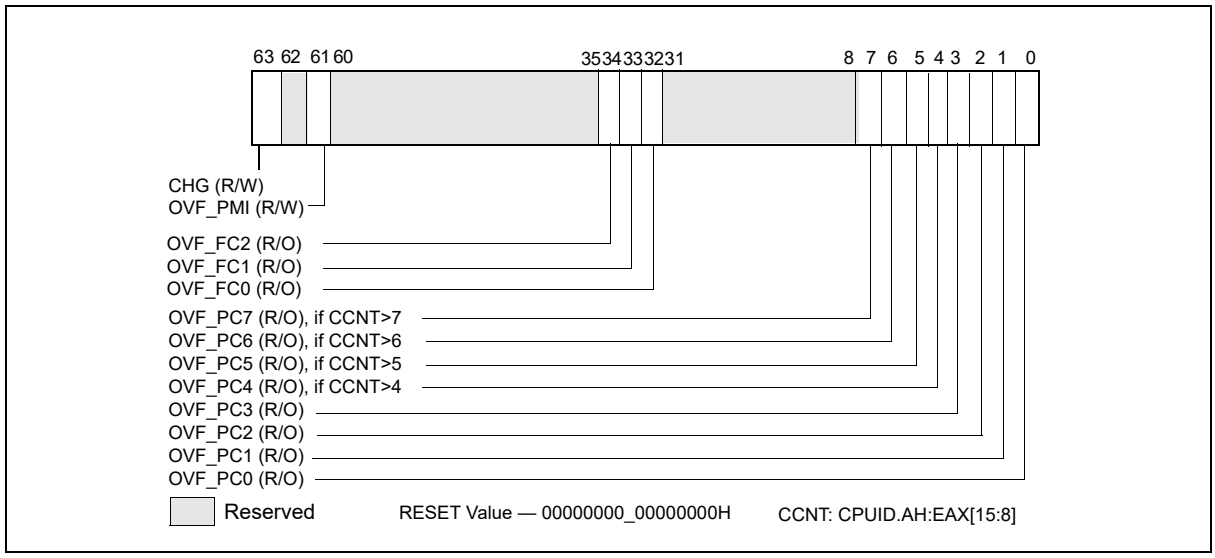


Figure 18-14. IA32_PERF_GLOBAL_STATUS MSR

18.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Intel micro-architecture code name Nehalem has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Intel microarchitecture code name Nehalem. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

18.3.1.1.1 Processor Event Based Sampling (PEBS)

All four general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 18-15.

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.

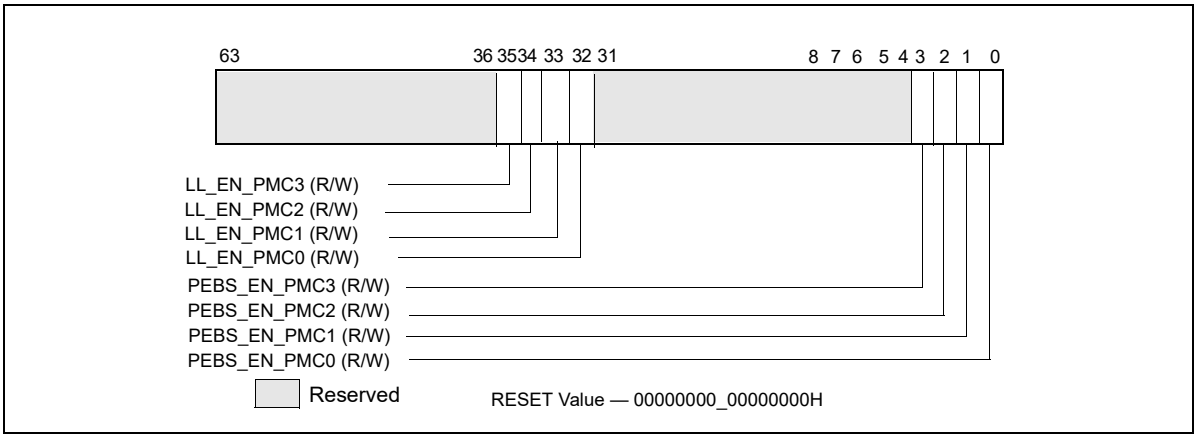


Figure 18-15. Layout of IA32_PEBS_ENABLE MSR

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
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding

Table 18-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
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 18-68. 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.

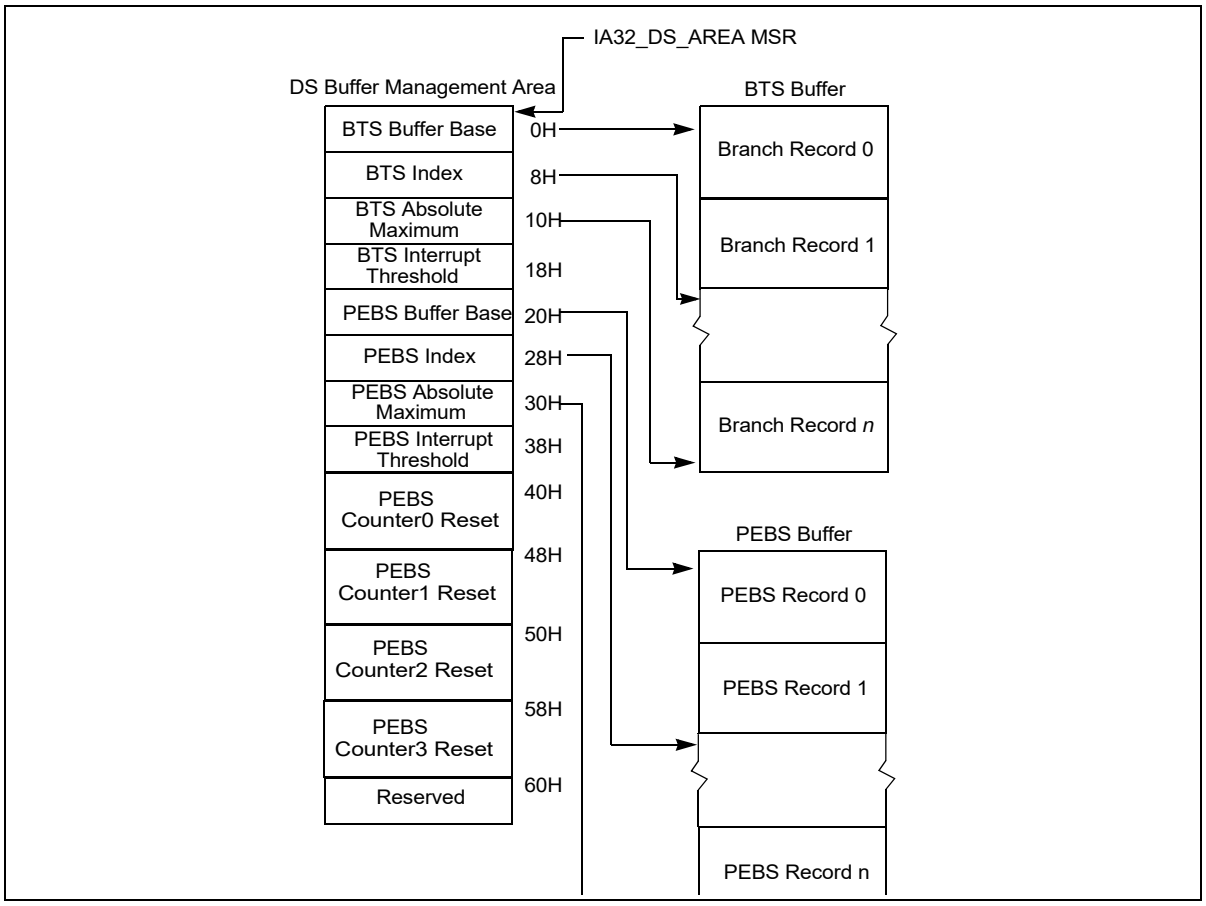


Figure 18-16. PEBS Programming Environment

- **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 was serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
07H[1]	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies found
08H	L3 MISS. Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation
0FH	The request was to un-cacheable memory.

NOTES:

1. Bit 7 is supported only for processor with CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 18-17.

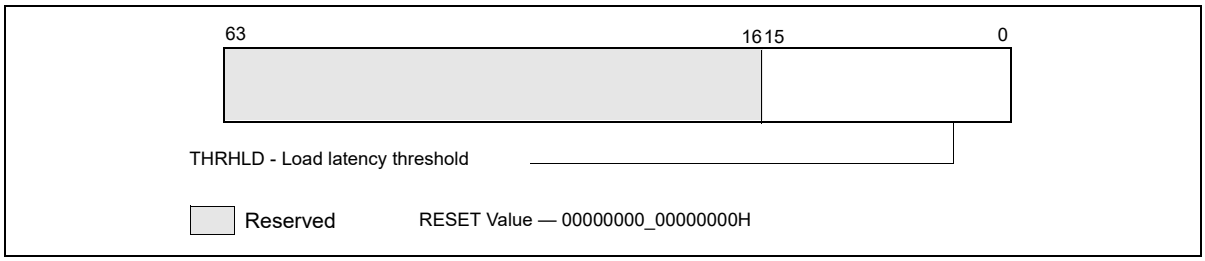


Figure 18-17. Layout of MSR_PEBS_LD_LAT MSR

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.

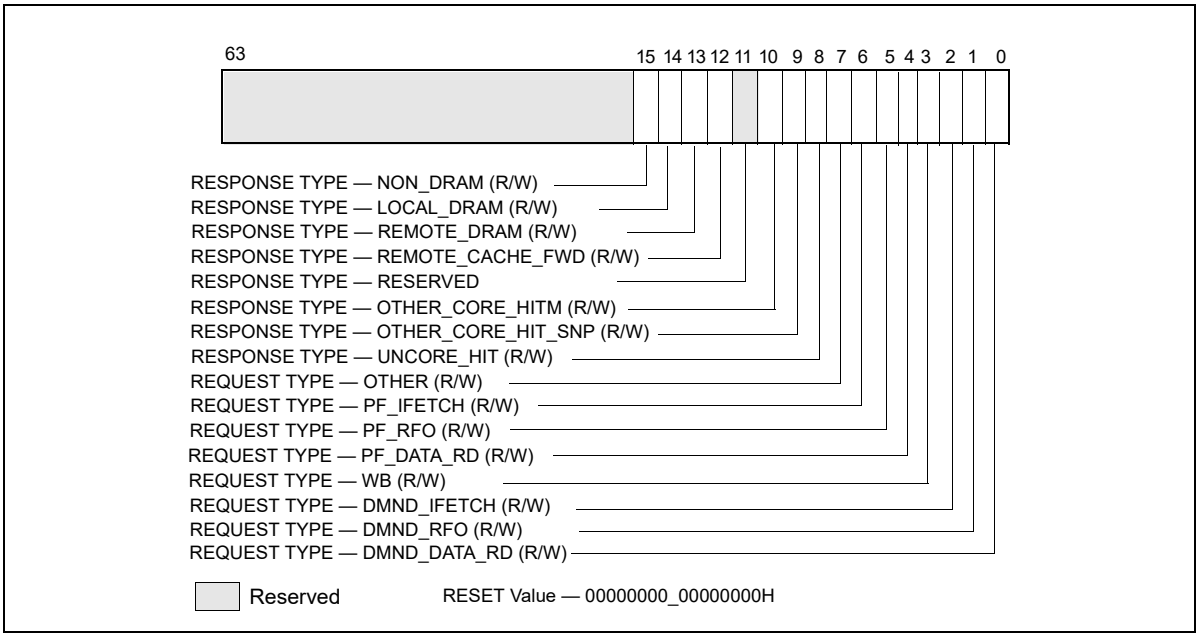


Figure 18-18. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
OTHER	7	(R/W). Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HITM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

18.3.1.2 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.

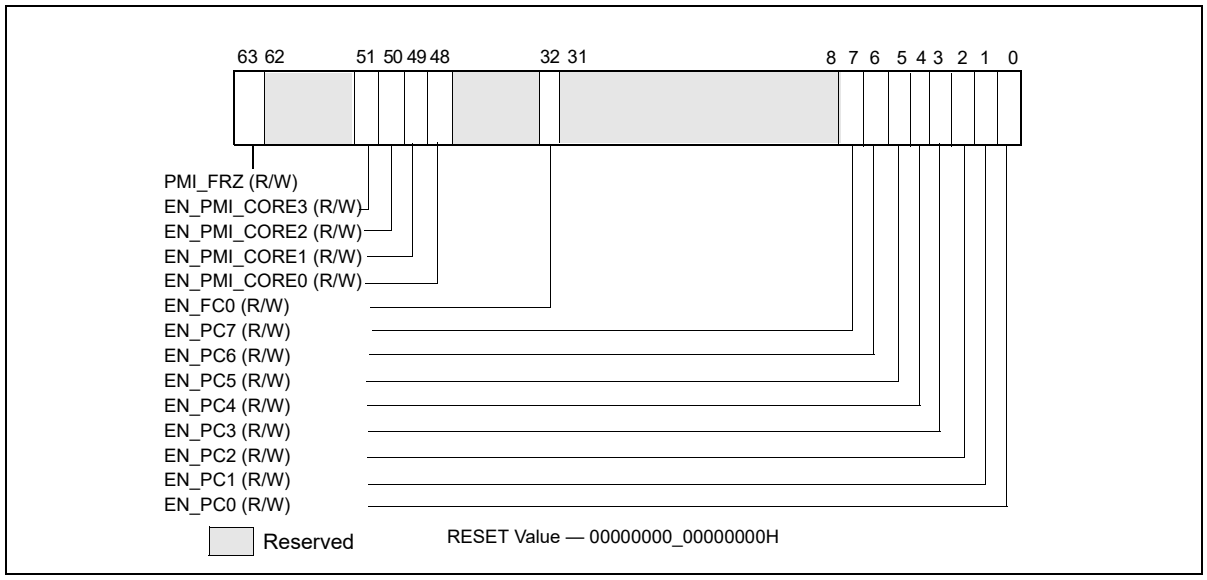


Figure 18-19. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

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.

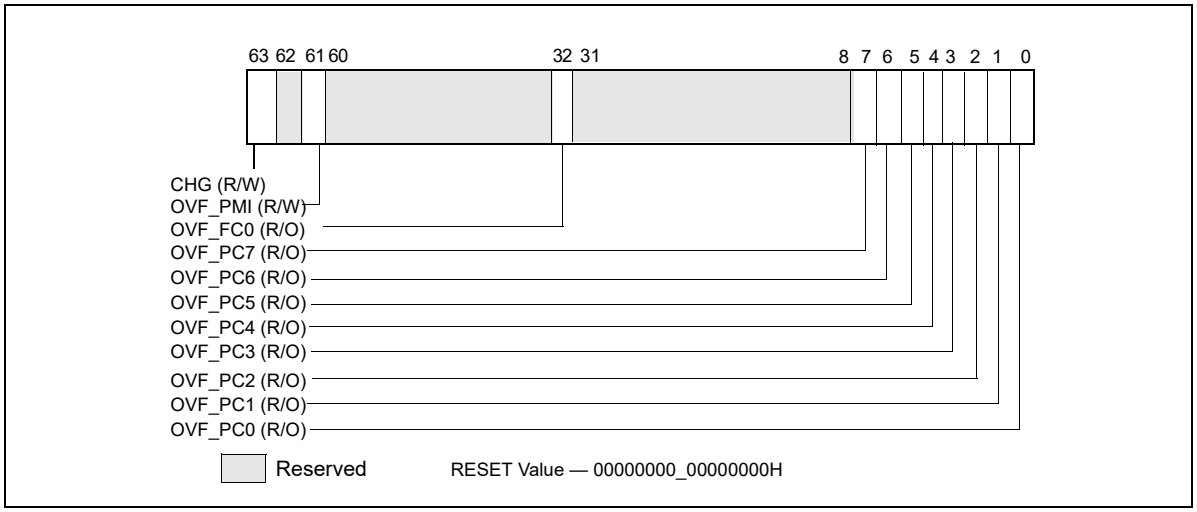


Figure 18-20. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 18-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

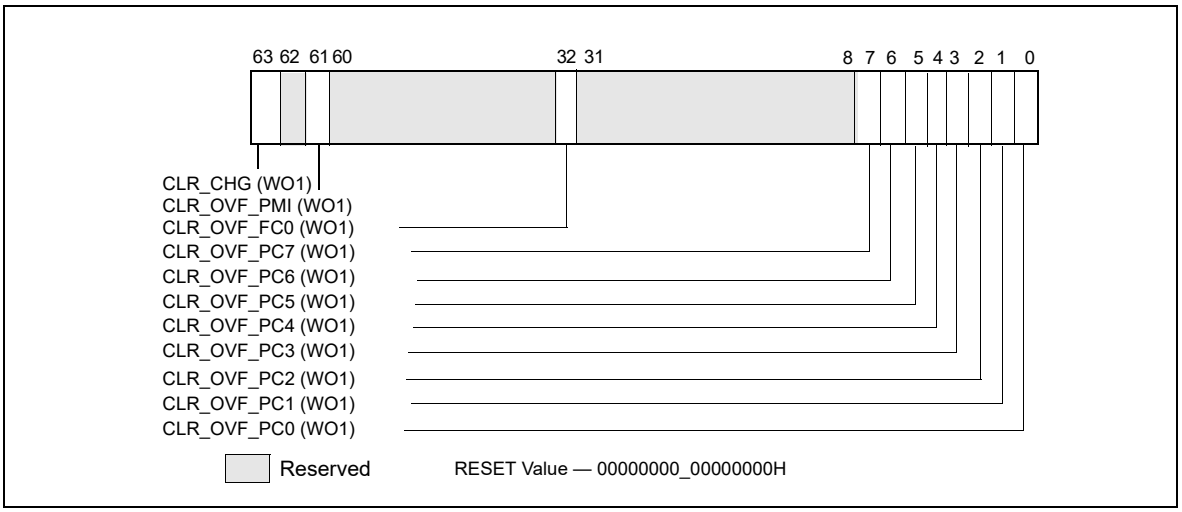


Figure 18-21. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- 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.

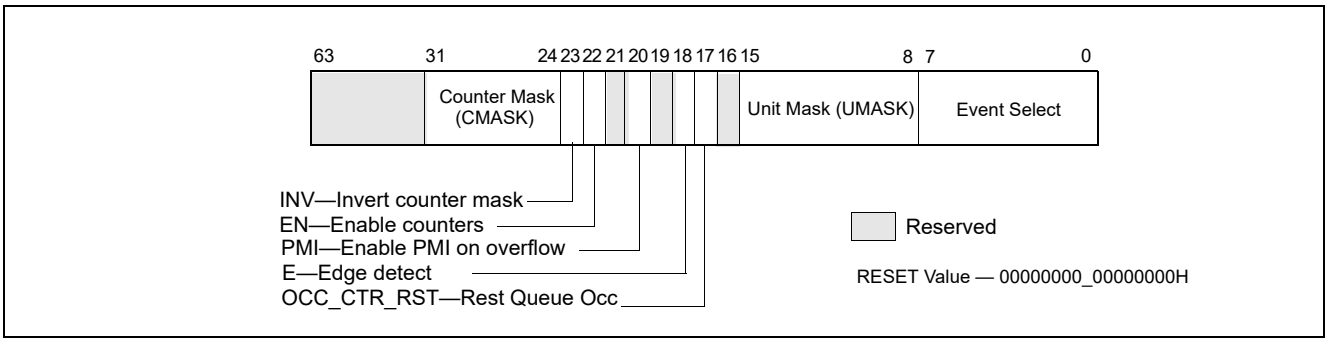


Figure 18-22. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- 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.

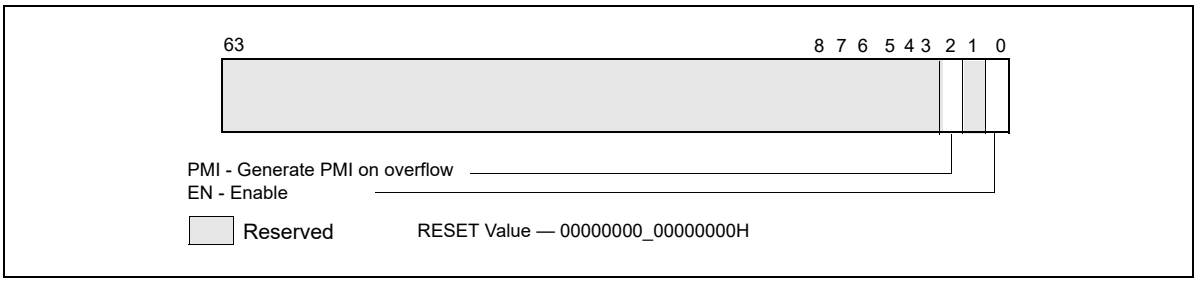


Figure 18-23. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- 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_PerfCntr) and the fixed-function counter (MSR_UNCORE_FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

18.3.1.2.3 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.

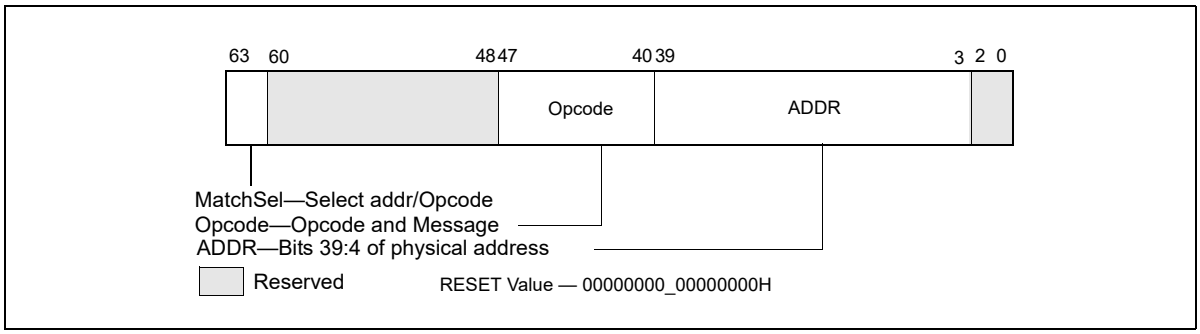


Figure 18-24. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- 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.

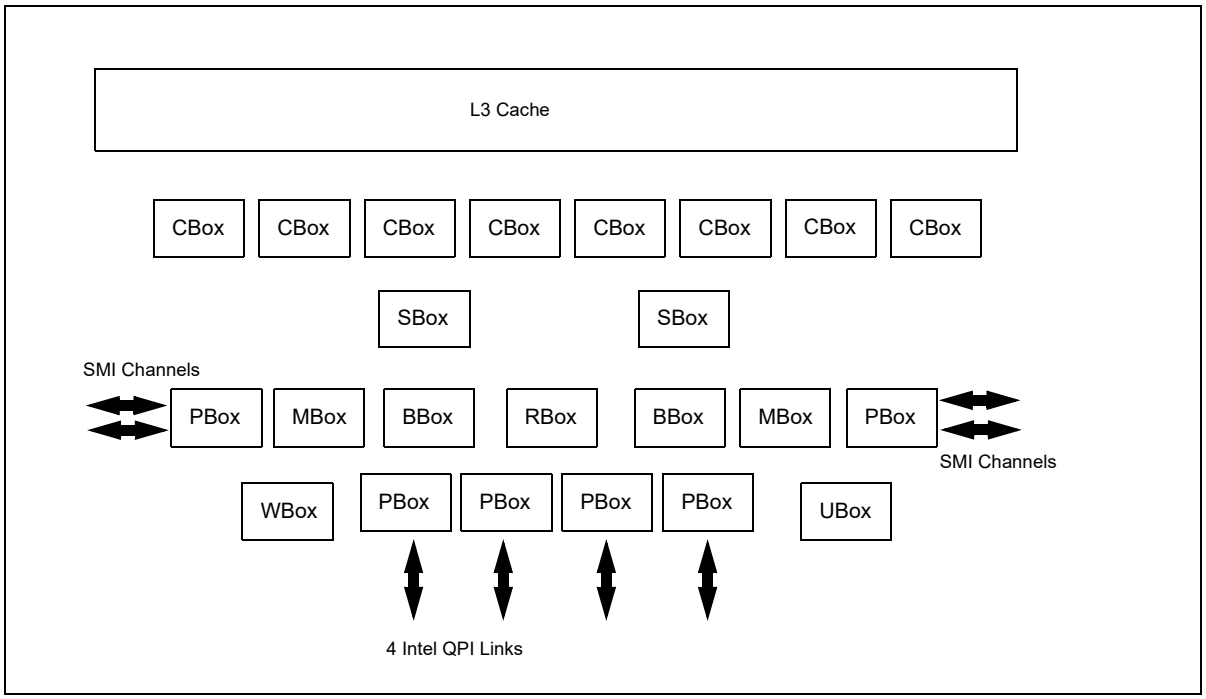


Figure 18-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

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 MSRs that provide uncore PMU interfaces are listed in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, Table 2-16 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more the one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_PMON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_CTR, MSR_S0_PMON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_EVNT_SEL, MSR_S0_PMON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR_M0_PMON_TIMESTAMP, MSR_R0_PMON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide".

18.3.2 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 19-29.

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 series[3]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 18-25, with the additional capability that up to 10 C-Box units are supported.

3. Exceptions are indicated for event code 0FH in Table 19-21; 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 enumerate # of counters.
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12.	See Table 18-68.	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; ▪ 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).

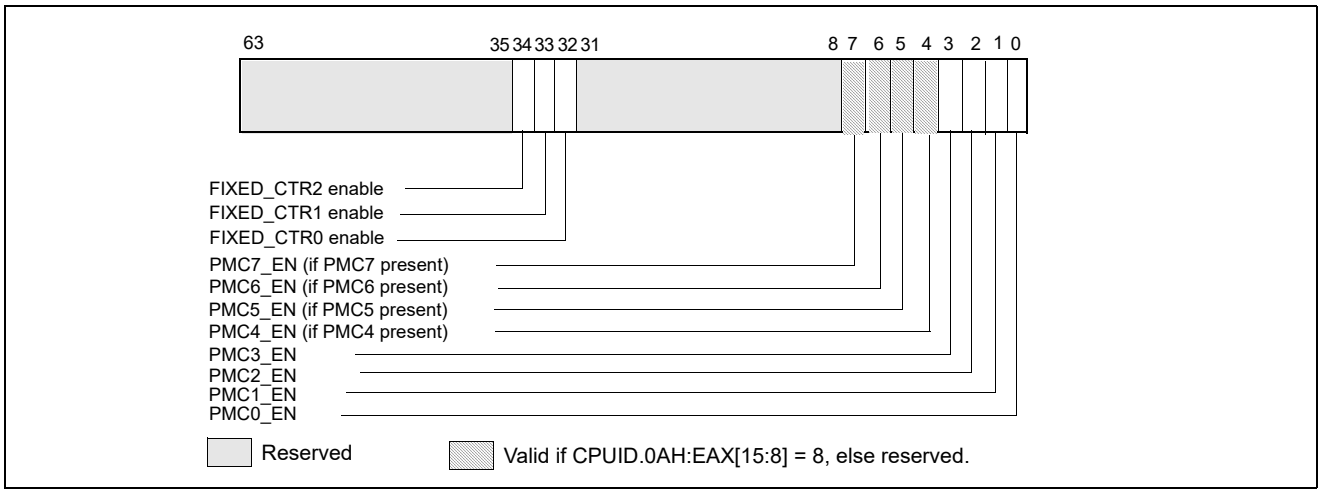


Figure 18-26. IA32_PERF_GLOBAL_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

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.

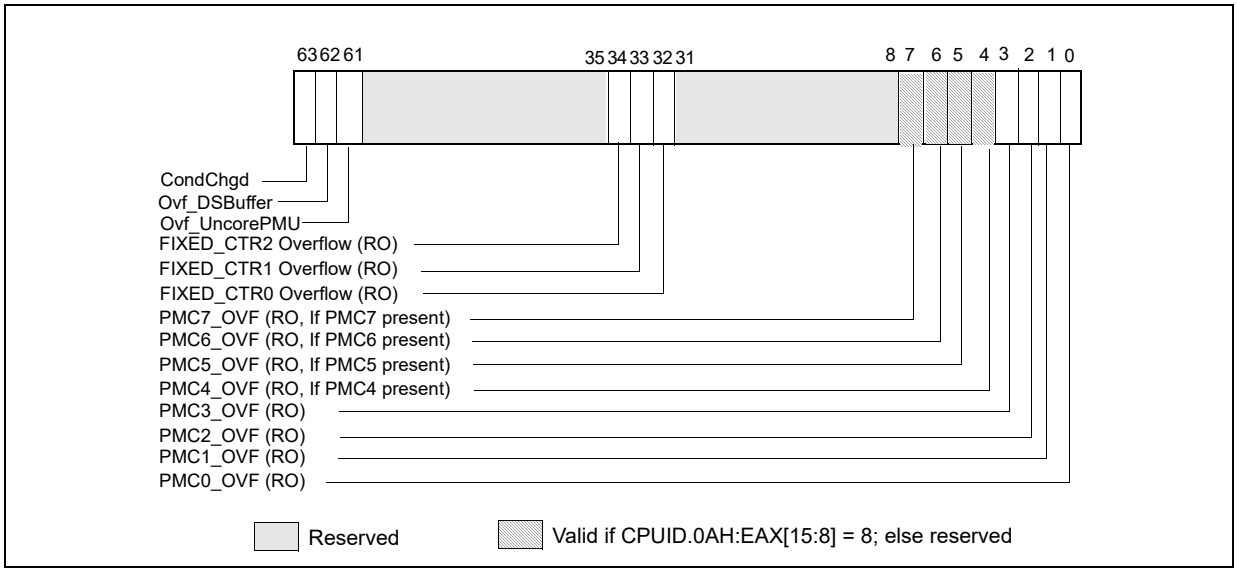


Figure 18-27. IA32_PERF_GLOBAL_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

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.

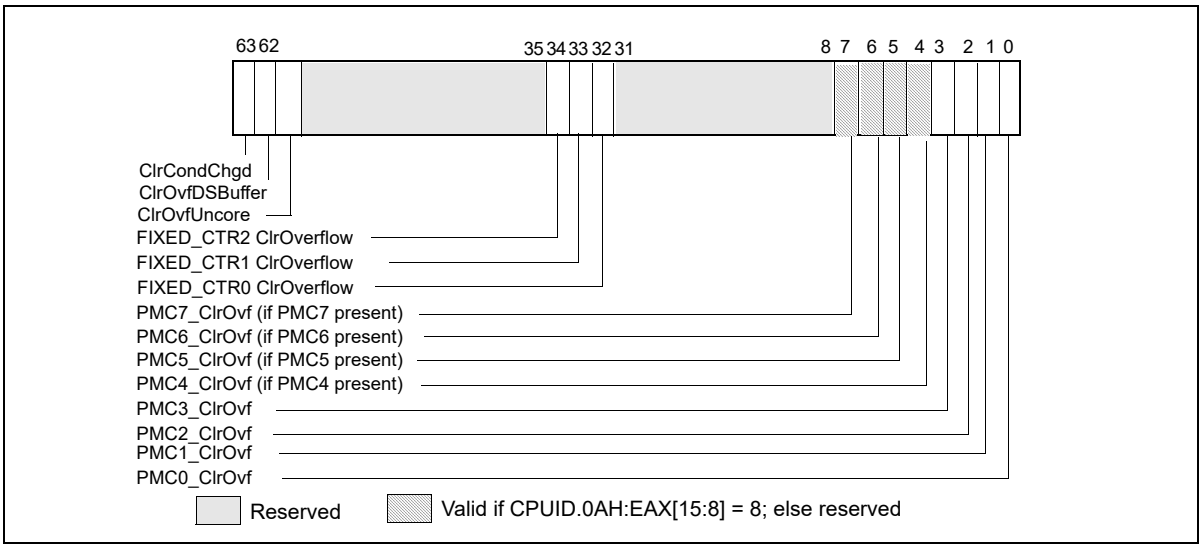


Figure 18-28. IA32_PERF_GLOBAL_OVF_CTRL MSR in Intel microarchitecture code name Sandy Bridge

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 either 4 or 8 depending specific processor's product features.

If a processor core is shared by two logical processors, each logical processors can access 4 counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, all eight general-purpose counters are visible, and CPUID.0AH:EAX[15:8] reports 8. IA32_PMC4-IA32_PMC7 occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_GLOBAL_OVF_CTL will also cause #GP.

18.3.4.3 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 18-68.	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.

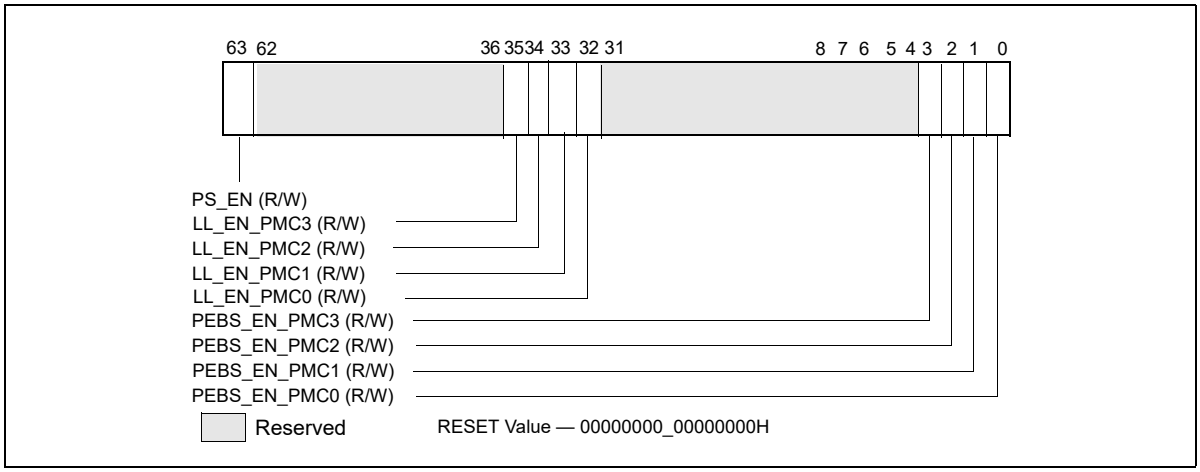


Figure 18-29. Layout of IA32_PEBS_ENABLE MSR

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	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

NOTES:
1. Only available on IA32_PMC1.

18.3.4.4.2 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_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-3. 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_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-3. 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 thus eliminating skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

18.3.4.5 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
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 18-30 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.

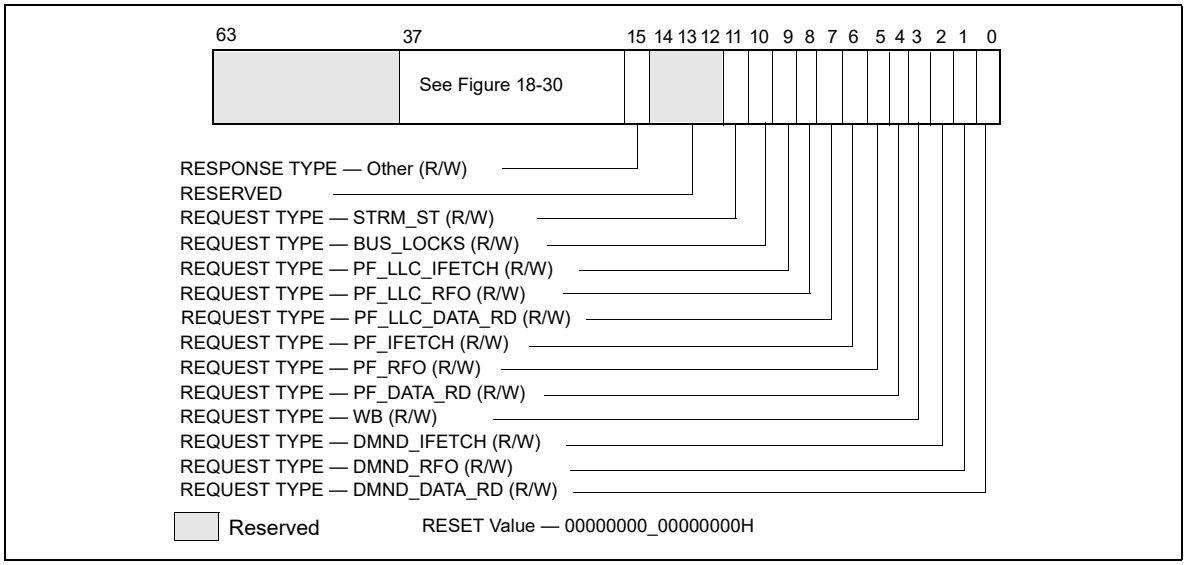


Figure 18-30. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 18-16. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	(R/W). L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	(R/W). RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	(R/W). L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

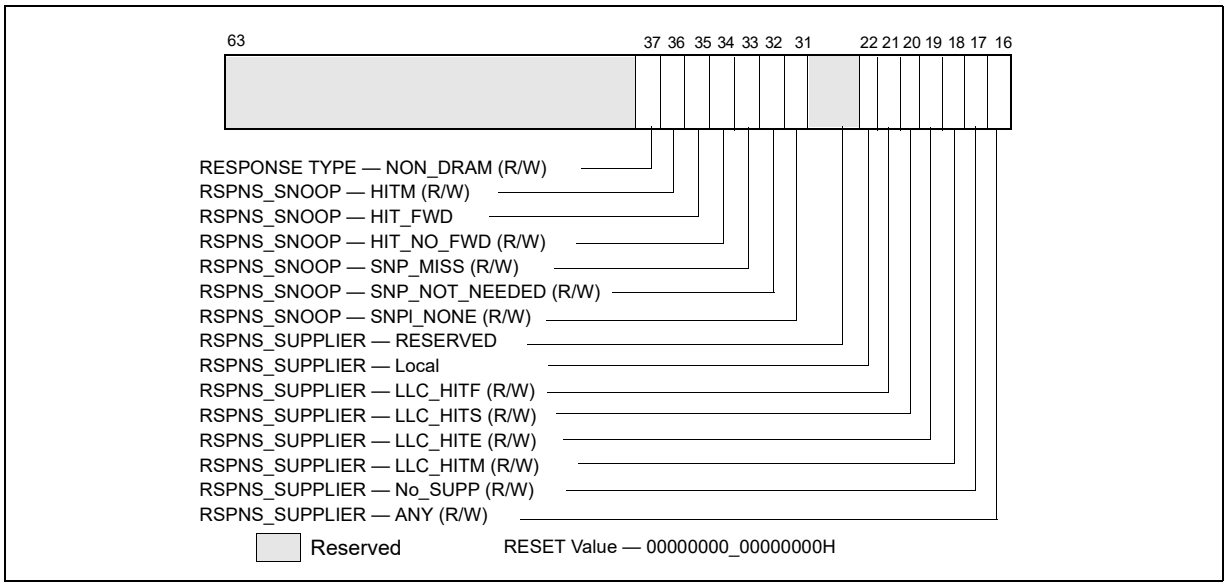


Figure 18-31. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

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	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY“ bit is set, the supplier and snoop info bits are ignored.

Table 18-18. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	SNP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitI was returned by all cores -For LLC Miss, RspI was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

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.

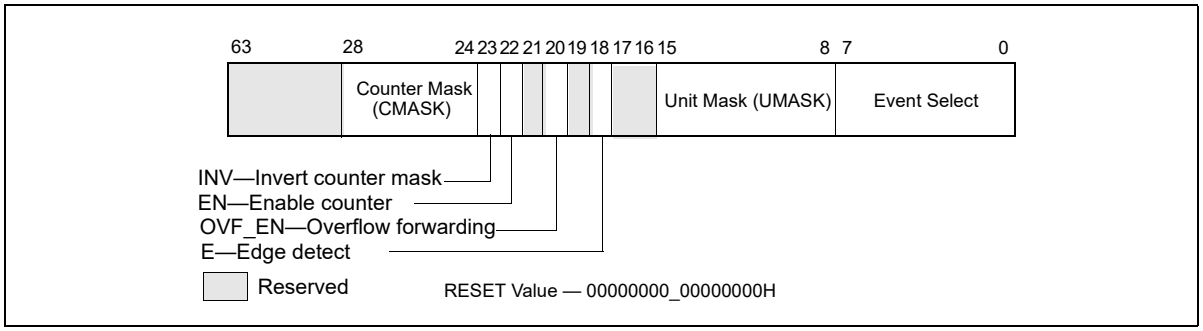


Figure 18-32. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see Table 19-18.
- 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).

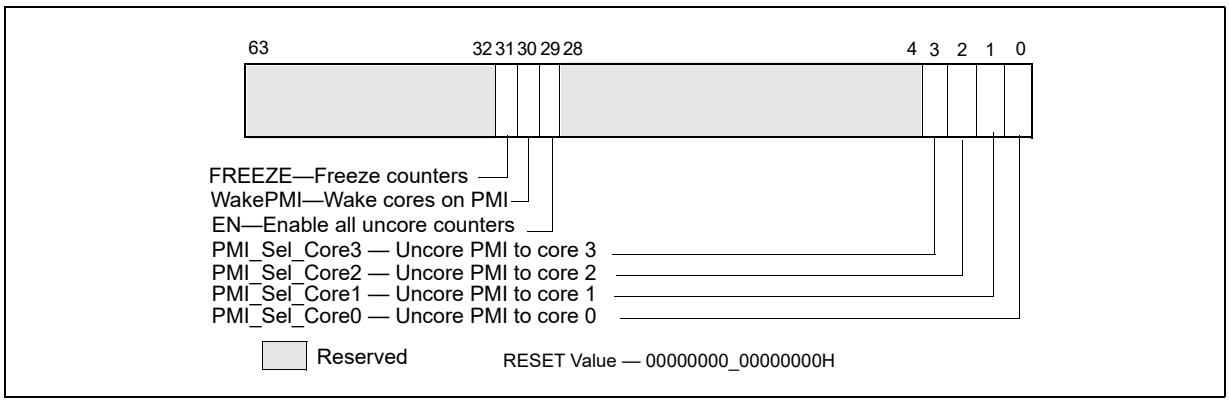


Figure 18-33. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs 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, seeTable 2-20 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 19-18 can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

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

18.3.4.7 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, the are some small differences in the non-architectural performance monitoring events (see Table 19-16).

Table 18-20. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Remote	30:23	(R/W): Remote DRAM Controller (either all 0s or all 1s)

18.3.4.8 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 2-23.

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 19-16.

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 2-27.

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	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12 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 microarchitecture code name Sandy Bridge, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Sandy Bridge is summarized in Table 18-23.

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 the an event listed in Table 18-25 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEBS_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

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	▪ **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	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
COREWB	3	(R/W). Counts the number of modified cachelines written back.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	(R/W). Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	(R/W). Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_ LOCK	10	(R/W). Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	(R/W). Counts the number of streaming store requests electronically.
Reserved	12-14	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-28. 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	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

Table 18-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	(R/W). L4 Cache
	Reserved	30:26	Reserved

18.3.6.4.1 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	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	L3_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	(R/W). Hop 0 Remote supplier
	L3_MISS_REMOTE_HOP1	28	(R/W). Hop 1 Remote supplier
	L3_MISS_REMOTE_HOP2P	29	(R/W). Hop 2 or more Remote supplier
	Reserved	30	Reserved

18.3.6.5 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 it's IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 18-34. 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 2-28.

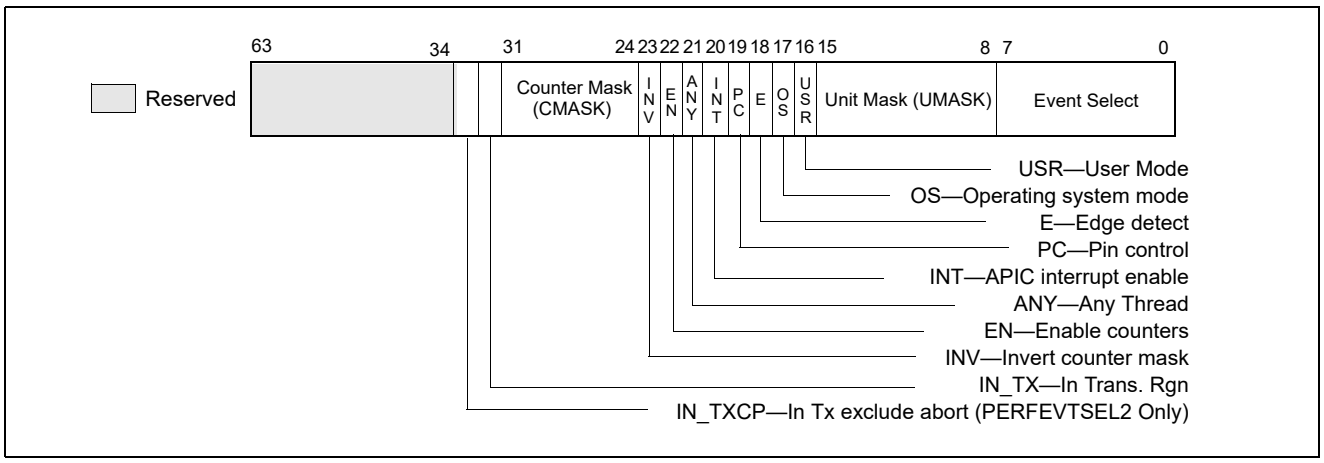


Figure 18-34. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

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. The following example illustrates using three counters to drill down cycles spent inside and outside of transactional regions:

- Program IA32_PERFEVTSEL2 to count Unhalted_Core_Cycles with (IN_TXCP=1, IN_TX=0), such that IA32_PMC2 will count cycles spent due to aborted TSX transactions;
- Program IA32_PERFEVTSEL0 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=1), such that IA32_PMC0 will count cycles spent by the transactional code regions;
- Program IA32_PERFEVTSEL1 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=0), such that IA32_PMC1 will count total cycles spent by the non-transactional code and transactional code regions.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they are listed in Table 19-10.

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 19-10 also support using PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORT ED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 18-31.

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, seeTable 2-20 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 19-11.

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 2-32.

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.

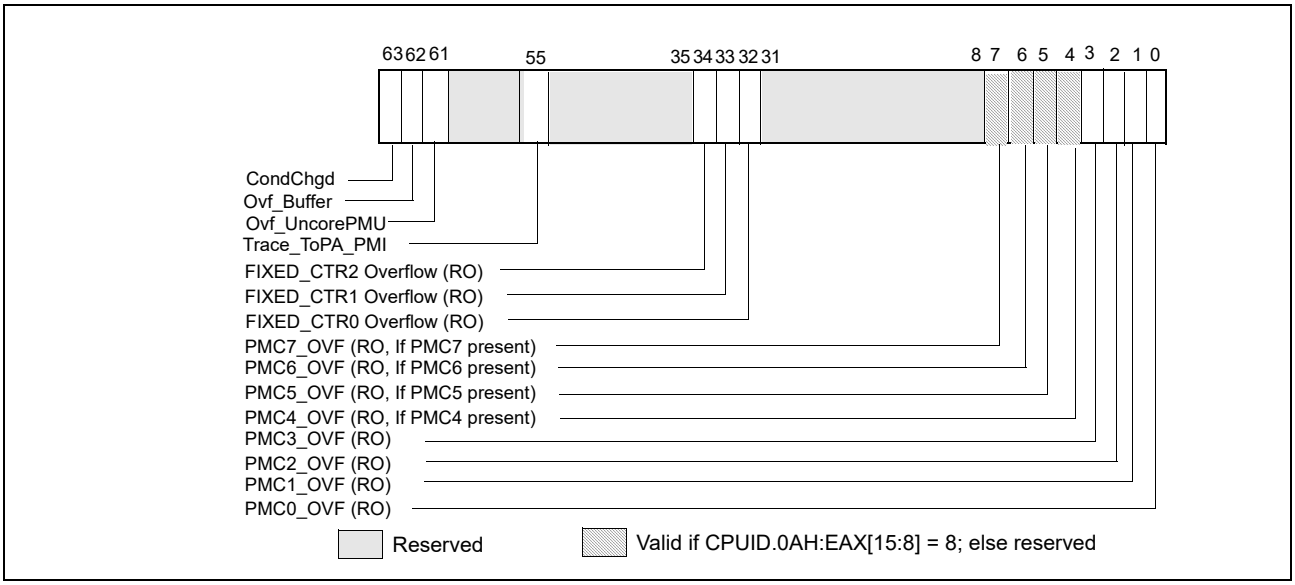


Figure 18-35. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

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.

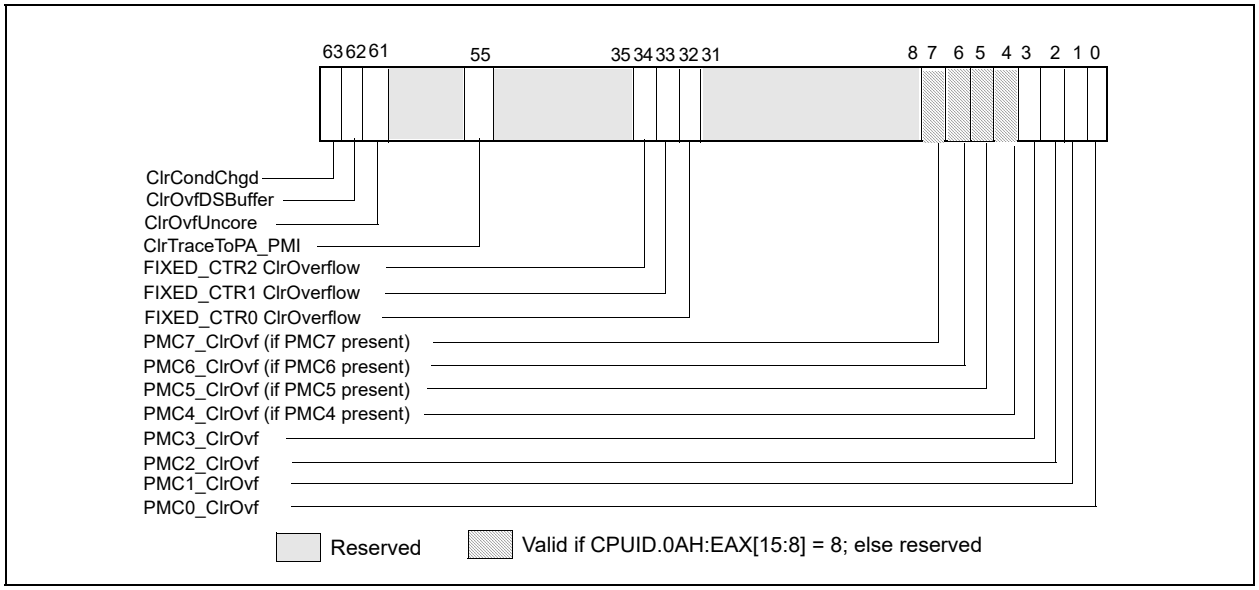


Figure 18-36. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, "Performance Monitoring Events".

18.3.8 6th Generation Intel® Core™ Processor and 7th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU's capability is similar to those described in Section 18.6.3 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 and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	CPUID enumerates # of counters.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_on_LBR with streamlined semantics. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz ▪ LBR_Frz	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-36.	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 and 7th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-34.

Table 18-34. PEBS Facility Comparison

Box	Intel® microarchitecture code name Skylake and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.3.1.1.1	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 and 7th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 18-35. 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 Intel Core Processor and 7th Generation Intel Core Processor Families

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
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 and 7th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS 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 and Kaby Lake microarchitectures is shown in Table 18-36.

Table 18-36. Precise Events for the Skylake and Kaby Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST[1]	01H
		ALL_CYCLES[2]	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable[3]>	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED[2]	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED[4]	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED[2]	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRED.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBS_FRONTEND, see Section 18.3.8.2
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 and 7th generation Intel Core processors is largely unchanged from prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

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	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES. ▪ Other bits are zero.
Reserved	A8H	Always zero.

18.3.8.1.4 PEBS Facility for Front End Events

In the 6th and 7th generation Intel Core processors, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The frontend microarchitectural conditions supported by PEBS requires the following interfaces:

- The IA32_PERFEVTSELx MSR must select "FrontEnd_Retired" (C6H) in the EventSelect field (bits 7:0) and umask = 01H,
- The "FRONTEND_RETIRED" event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 18-38.
- 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.
L1I_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss[1]. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.

NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 18-39.

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) Frontend events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a frontend (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

18.3.8.2 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 30:16): see Table 18-41.
- Snoop response information (bits 37:31): see Table 18-42.

Table 18-40. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake and Kaby Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	6:3	Reserved
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC prefetches into L3.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	10:9	Reserved
STRM_ST	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-41 lists the supplier information field that applies to 6th and 7th generation Intel Core processors. (6th generation Intel Core processor CPUID signature: 06_4EH, 06_5EH; 7th generation Intel Core processor CPUID signature: 06_8EH, 06_9EH).

Table 18-41. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_4EH, 06_5EH and 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available.
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT	22	(R/W). L4 Cache (if L4 is present in the processor)
	Reserved	25:23	Reserved
	DRAM	26	(R/W). Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	(R/W). L4 cache super line hit (if L4 is present in the processor)

Table 18-42 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	SNOOP_NONE	31	(R/W). No details on snoop-related information
	SNOOP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNOOP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitI was returned by all cores -For LLC Miss, RspI was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

18.3.8.2.1 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 30:16): see Table 18-44.
- Snoop response information has not been changed and is the same as in (bits 37:31): 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	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	(R/W). Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	(R/W). Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	(R/W). Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
STREAMING_STORES	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved.
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-44 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	(R/W). Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	(R/W). No Supplier Information available.
	L3_HIT_M	18	(R/W). M-state initial lookup stat in L3.
	L3_HIT_E	19	(R/W). E-state
	L3_HIT_S	20	(R/W). S-state
	L3_HIT_F	21	(R/W). F-state
	Reserved	25:22	Reserved.
	L3_MISS_LOCAL_DRAM	26	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	(R/W). Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	(R/W). Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	(R/W). Hop 2 or more Remote supplier.
	Reserved	30	Reserved.

18.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

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.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3 in the SDM. The processor supports AnyThread counting in three architectural performance monitoring events.

18.4.1.1 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 18-45. 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
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 18-46, and each field in the PEBS record is 64 bits long.

Table 18-46. 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 18-47 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 18-47. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 18-48. 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 18-48. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		15	ANY_REQUEST	Account for any requests.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.

Table 18-48. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
		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.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMC0. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMC0. This bit is reserved for OFFCORE_RESP_1 event.

18.4.1.1.3 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).

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 19-29.

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 18-61, Table 18-62, Table 18-63, and Table 18-64. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-29 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 19-28.

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 18-49.

Table 18-49. 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 18-50, and each field in the PEBS record is 64 bits long.

Table 18-50. 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

Table 18-50. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
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 18-51 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 18-51. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 18-37 and Figure 18-38. 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.

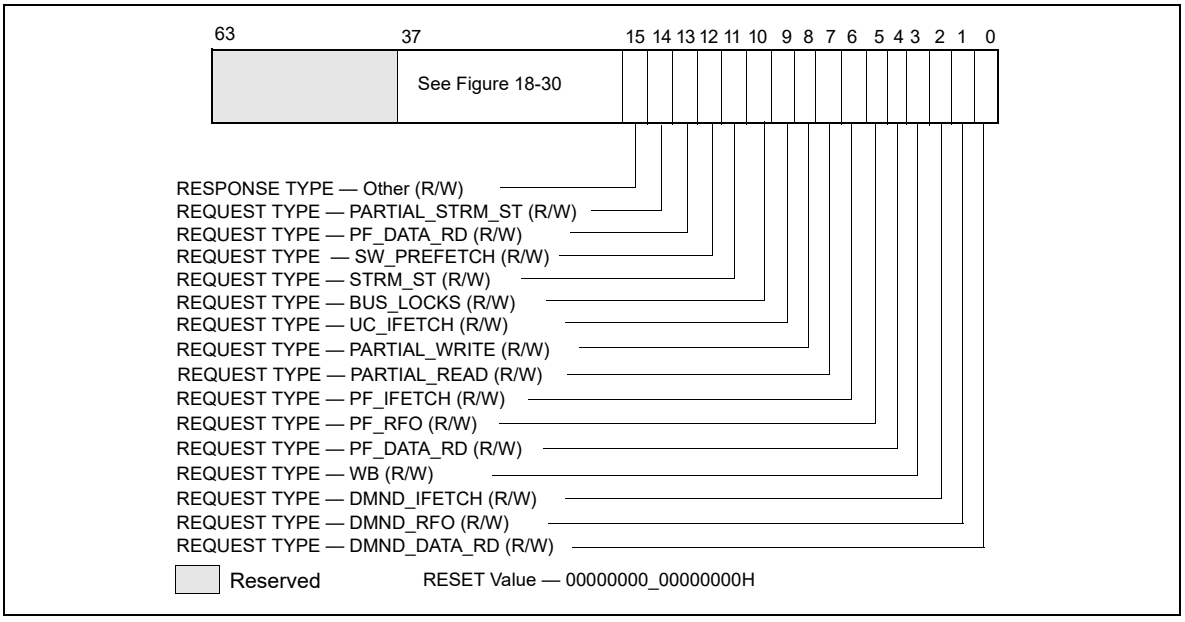


Figure 18-37. Request_Type Fields for MSR_OFFCORE_RSPx

Table 18-52. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	(R/W). Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	(R/W). Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	(R/W). Counts the number of UC instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
SW_PREFETCH	12	(R/W). Counts software prefetch requests
PF_DATA_RD	13	(R/W). Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	(R/W). Streaming store requests
ANY	15	(R/W). Any request that crosses IDI, including I/O.

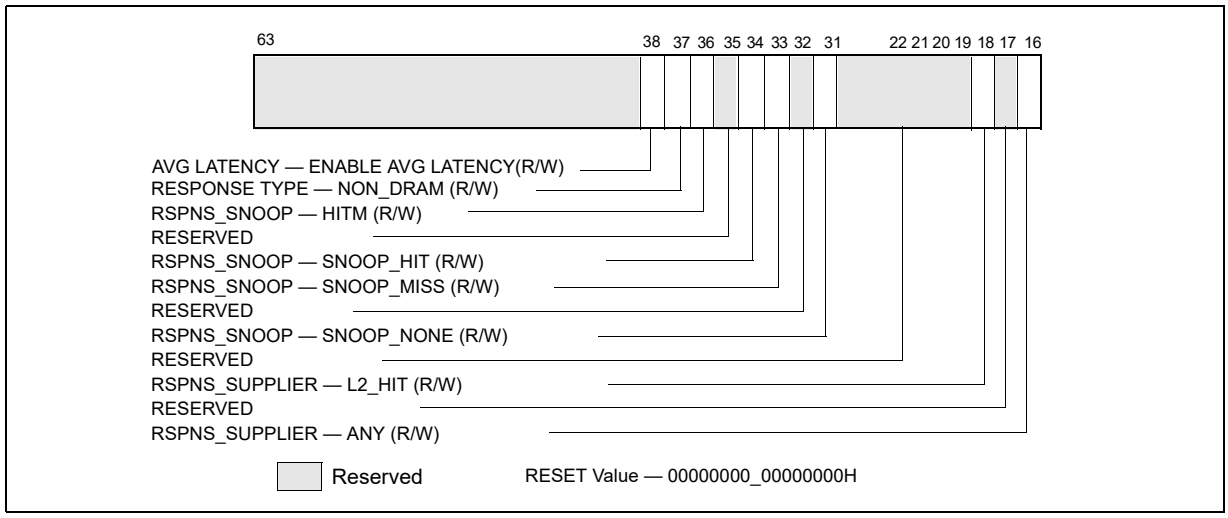


Figure 18-38. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 18-52) and a valid response type pattern (Table 18-53, Table 18-54). 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 18-53. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	(R/W). Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	(R/W). Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

Table 18-54. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	(R/W). Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	(R/W). Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved
	HITM	36	(R/W). Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	(R/W). Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0). This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

18.5.2.3 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_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, white setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be

obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

18.5.3 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 18-55.

Table 18-55. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	4	2	
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 18.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.	▪ 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	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz	▪ Individual counter overflow ▪ PEBS buffer overflow	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	No	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 18.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-56.	See Section 18.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-49).	IA32_PMC0 only.

Table 18-55. 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 18-57; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	Table 18-50.	
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 18-56.

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 over-flows, 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 18-56. Precise Events Supported by the Goldmont Microarchitecture

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	C0H	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Intel Goldmont microarchitecture is shown in Table 18-57, and each field in the PEBS record is 64 bits long.

Table 18-57. 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_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g. via

IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g. what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

18.5.3.1.3 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 18-51 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 18-58.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-53.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-59.
- 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 18-58. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	(R/W) Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	(R/W) Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	(R/W) Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	(R/W) Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	(R/W) Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	(R/W) Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	(R/W) Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	(R/W) Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	(R/W) Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	(R/W) Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	(R/W) Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	(R/W) Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	(R/W) Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	(R/W) Counts partial cacheline writes due to streaming stores.

Table 18-58. MSR_OFFCORE_RSPx Request_Type Field Definition (Contd.)

Bit Name	Offset	Description
ANY_REQUEST	15	(R/W) Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-52) and a valid response type pattern (either Table 18-53 or 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 18-59. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	(R/W). A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	(R/W) A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	(R/W) Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	(R/W) Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests[1]	OUTSTANDING	38	(R/W) Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

NOTES:

1. See Section 18.5.2.3, "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 'OR' (L2 Hit)] 'XOR' (Snoop Info Bits) 'XOR' (Avg Latency)

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 18-60.

Table 18-60. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	No change.
# of general-purpose counters per core	4	4	No change.
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 18-56.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

18.5.4.1 Extended PEBS

The Extended PEBS feature, introduced in Goldmont Plus microarchitecture, supports PEBS (Processor Event Based Sampling) on a fixed-function performance counters as well as all four general purpose counters (PMC0-3). PEBS can be enabled for the four general purpose counters using PEBS_EN_PMCi bits of IA32_PEBS_ENABLE (i = 0, 1, 2, 3). PEBS can be enabled for the 3 fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PEBS_ENABLE (I = 0, 1, 2).

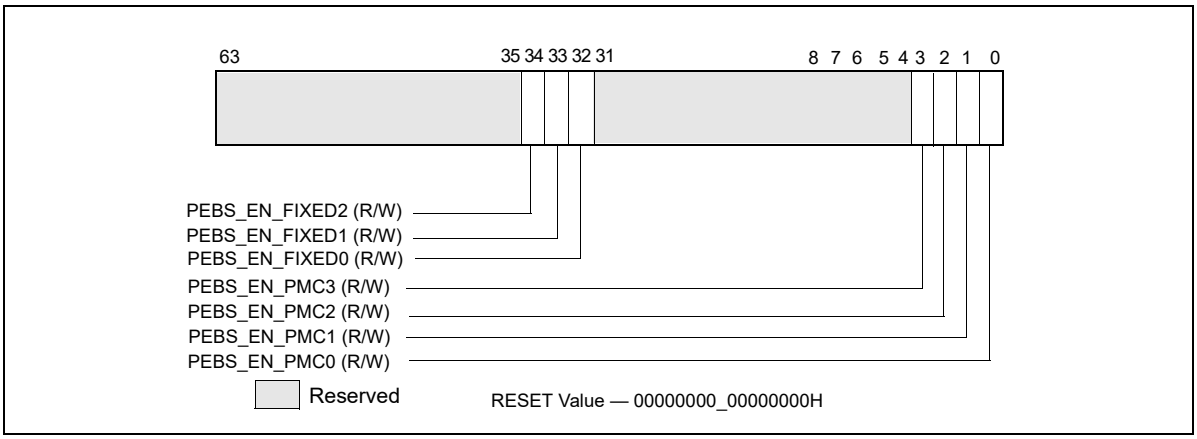


Figure 18-39. Layout of IA32_PEBS_ENABLE MSR

Similar to Goldmont microarchitecture, Goldmont Plus microarchitecture processors can generate PEBS record events on both precise as well as non-precise events.

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.

IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1 counts unhalted core cycles and is a non-precise event. IA32_FIXED_CTR2 counts unhalted reference cycles and is a non-precise event.

The Applicable Counter field at offset 90H 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 for Goldmont Plus is shown in Figure 18-40. 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).

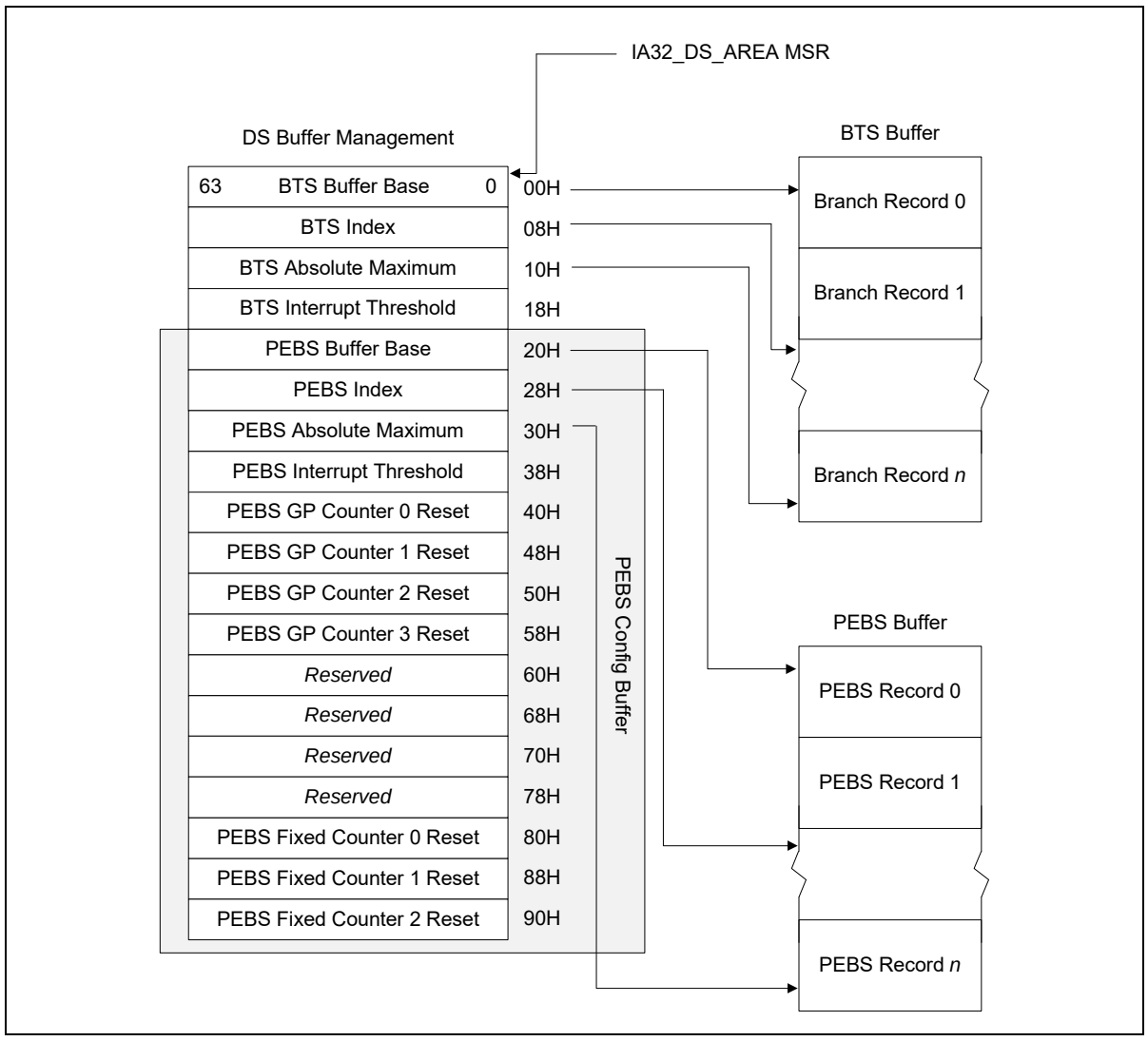


Figure 18-40. PEBS Programming Environment

18.5.4.2 Reduced Skid PEBS

Goldmont Plus microarchitecture processors supports the Reduced Skid PEBS feature described in Section 18.5.3.1.2 on the IA32_PMC0 counter. Although Goldmont Plus adds support for generating PEBS records for precise events on the other general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

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 18-61. 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 18-61. 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 18-62.

Table 18-62. 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 18-63.

Table 18-63. 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 18-64. 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 18-64. 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 18-61, Table 18-62, Table 18-63, and Table 18-64. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-25 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 18-65.

Table 18-65. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	HITM response
Bit 10	Reserved

Table 18-65. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
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 18-66.

Table 18-66. 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 MSR_PERF_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 18-41. 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.

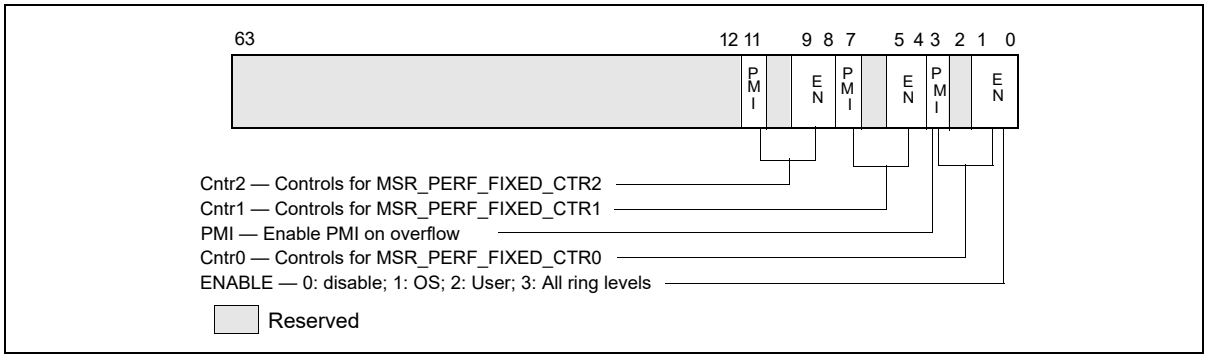


Figure 18-41. Layout of MSR_PERF_FIXED_CTR_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

18.6.2.2 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 (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 18-42). 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 MSR_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

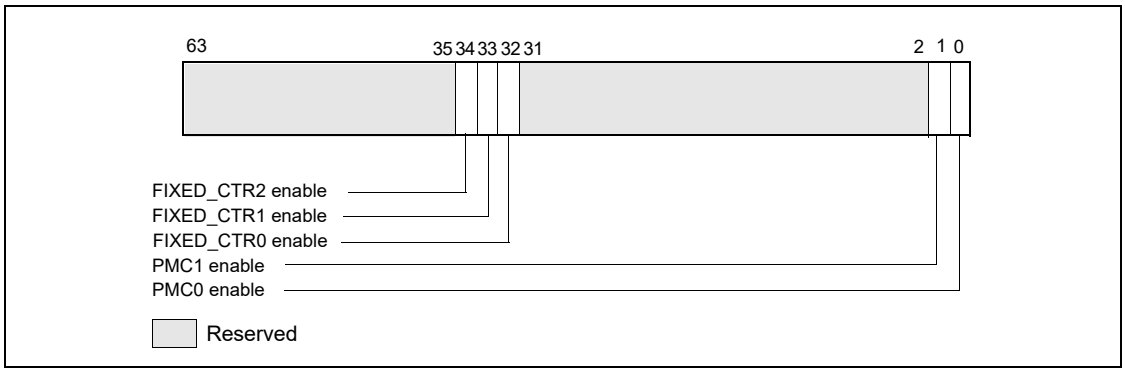


Figure 18-42. Layout of MSR_PERF_GLOBAL_CTRL MSR

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.

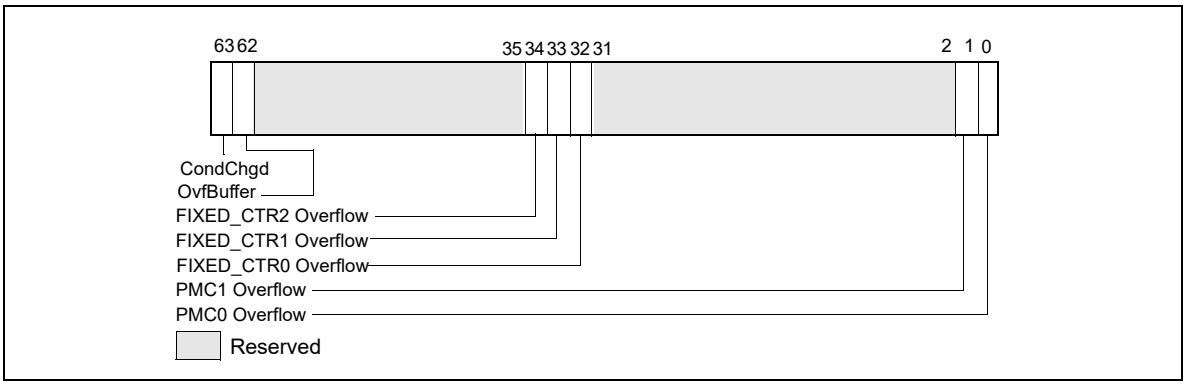


Figure 18-43. Layout of MSR_PERF_GLOBAL_STATUS MSR

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.

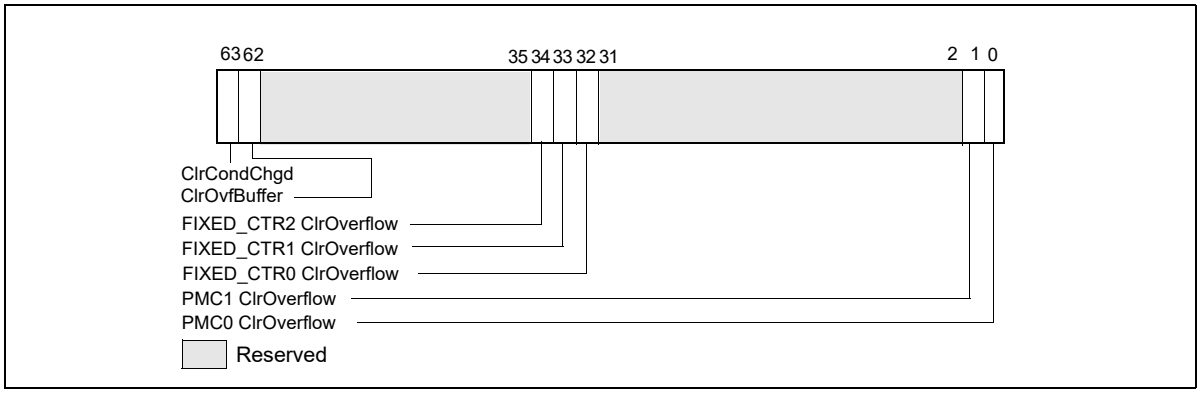


Figure 18-44. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

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 micro-architecture does not require special MSR programming control (see Section 18.6.3.6, "At-Retirement Counting"), but is limited to IA32_PMC0. See Table 18-67 for a list of events available to processors based on Intel Core micro-architecture.

Table 18-67. 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 18-68. The procedure for detecting availability of PEBS is the same as described in Section 18.6.3.8.1.

Table 18-68. 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 18-68.

18.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of

IA32_PERF_CAPABILITES are defined in Table 2-2 of Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1
- PEBSRecordFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (seeSection 18.6.3.8).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (seeSection 18.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (seeSection 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).

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 18-69.

Table 18-69. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS.	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled.	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: ▪ Clear all counters if "Counter Freeze on PMI" is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	

Table 18-69. Requirements to Program PEBS (Contd.)

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

NOTES:

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

18.6.2.4.4 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 18-70 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, "Perfor-

mance Monitoring Events."

Table 18-70. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H

Table 18-70. 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_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH

NOTES:

1. MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events (see Table 19-31) are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events (see Table 19-32) are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

  The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting —** A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling —** A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- **Processor event-based sampling (PEBS) —** In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

18.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-70) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-70) 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.)

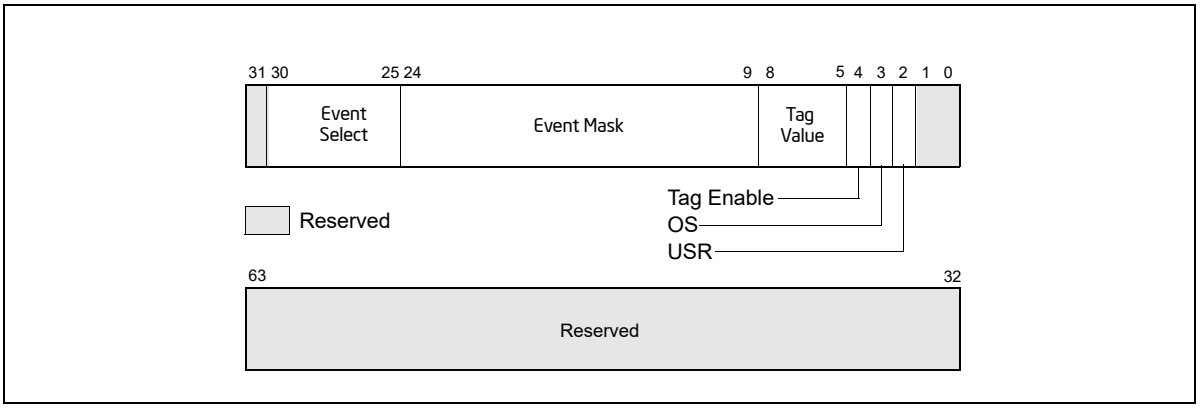


Figure 18-45. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support

- **Tag enable, bit 4 —** When set, enables tagging of μops 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 μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-70 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 18-70). 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 the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

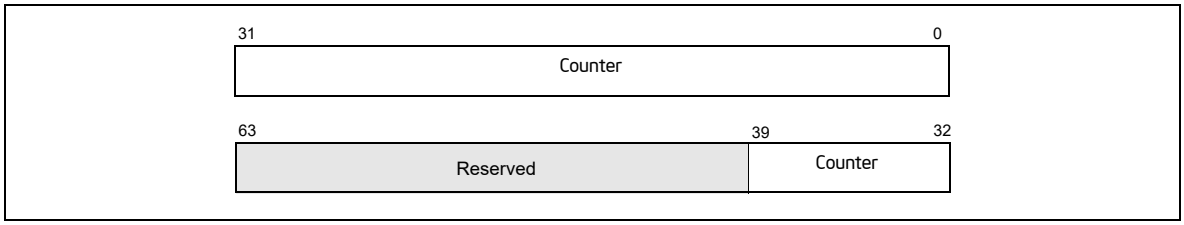


Figure 18-46. Performance Counter (Pentium 4 and Intel Xeon Processors)

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 18-70). 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.

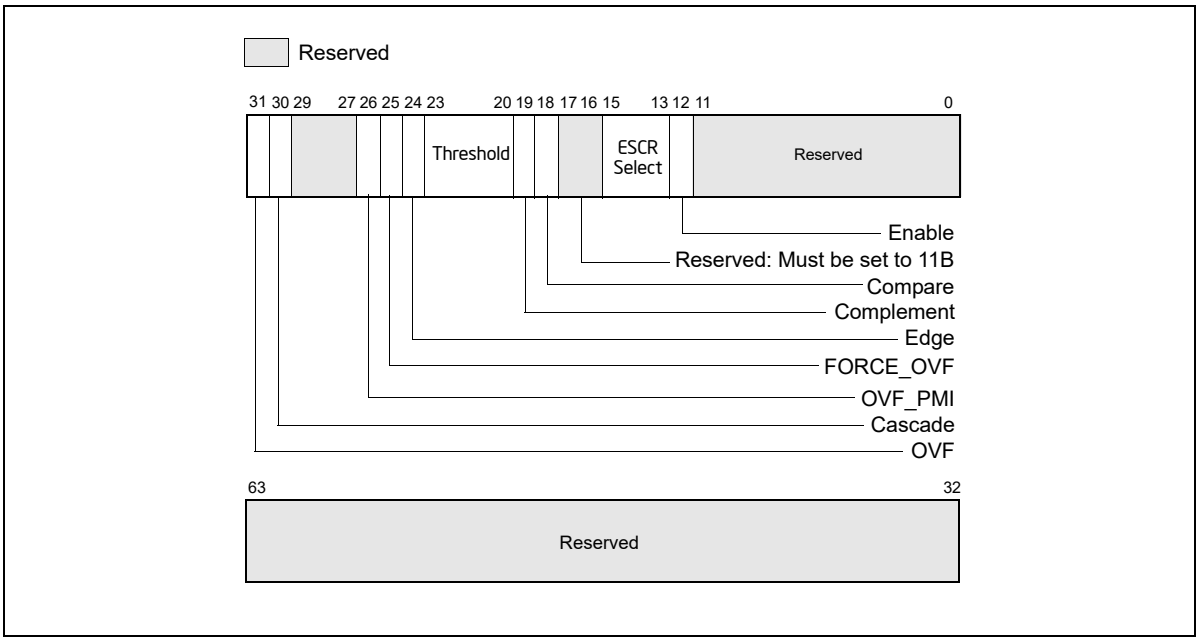


Figure 18-47. Counter Configuration Control Register (CCCR)

- **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 19-31, Chapter 19.
3. Match the CCCR Select value and ESCR name in Table 19-31 to a value listed in Table 18-70; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which the are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

18.6.3.5.1 Selecting Events to Count

Table 19-32 in Chapter 19 lists a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event listed in Table 19-32, setup information is provided. Table 18-71 gives an example of one of the events.

Table 18-71. 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]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

For Table 19-31 and Table 19-32, 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 18-70 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 18-70.
- **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 19-32.)
- **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 19-32.)

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 19-31, 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 18-70.
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 counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 18-48 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."

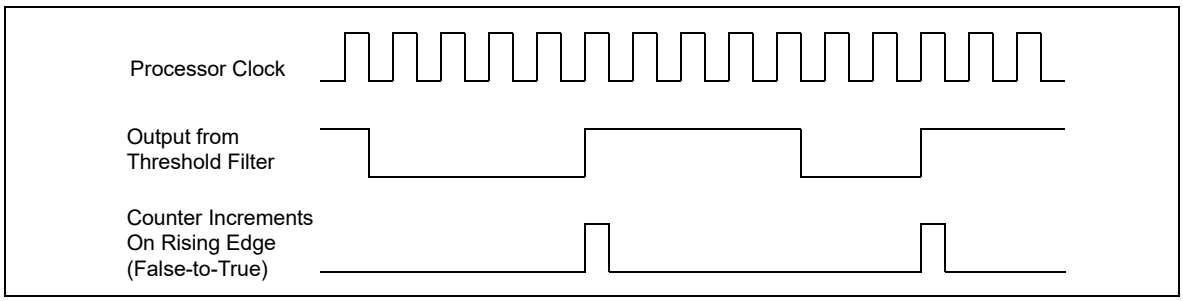


Figure 18-48. Effects of Edge Filtering

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 18-70 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 18-70). 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 18-72.

Table 18-72. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1|2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INTO0	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INTO3	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INTO4	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INTO5	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 18-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

18.6.3.5.8 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 19-32 through 19-36 list predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire —** In at-retirement event descriptions, the term "bogus" refers to instructions or μops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or μops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction_Retired and Uops_Retired in Table 19-32) can count instructions or μops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging —** Tagging is a means of marking μops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per μop and a direct count of the event would not provide an indication of how many μops encountered that event.

  The tagging mechanisms allow a μop to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per μop, rather than once per

event. For example, a μop may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired μops that encountered a cache miss) will increment only once for that μop. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, "Related Literature").

- **Replay —** To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist —** When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μops before they begin and are costly.

18.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-32, a μop may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count μops tagged with that mechanism:

- **Front-end tagging —** This mechanism pertains to the tagging of μops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event.
- **Execution tagging —** This mechanism pertains to the tagging of μops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.
- **Replay tagging —** This mechanism pertains to tagging of μops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.
- **No tags —** This mechanism does not use tags. It uses the Instr_retired and the Uops_ retired events.

Each tagging mechanism is independent from all others; that is, a μop that has been tagged using one mechanism will not be detected with another mechanism's tagged-μop detector. For example, if μops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged μops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of μops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

Table 19-32 lists the performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag μop and count tagged μops.

18.6.3.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts μops that have been tagged as encountering any of the following events:

- **μop decode events —** Tagging μops for μop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.
- **Trace cache events —** Tagging μops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR (see Table 19-34).

Table 19-32 describes the Front_end_event and Table 19-34 describes metrics that are used to set up a Front_end_event count.

The MSRs specified in the Table 19-32 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 19-32 describes the Execution_event and Table 19-35 describes metrics that are used to set up an Execution_event count.

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect μops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular μop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged μops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. This mechanism is summarized in the Table 19-35 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 19-32 describes the Replay_event and Table 19-36 describes metrics that are used to set up an Replay_event count.

The replay mechanism enables tagging of μops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μop that may experience the replay in the MSR_PEBS_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBS_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBS_ENABLE MSR.

The Table 19-36 lists the metrics that are support the replay tagging mechanism and the at-retirement events that use the replay tagging mechanism, and specifies how the appropriate MSRs need to be configured. The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 17.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE_MSR. Each of these metrics requires that the Replay_Event (see Table 19-32) be used to count the tagged μops.

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_PERFEVTSEL0 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 19-32 through 19-36.

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.

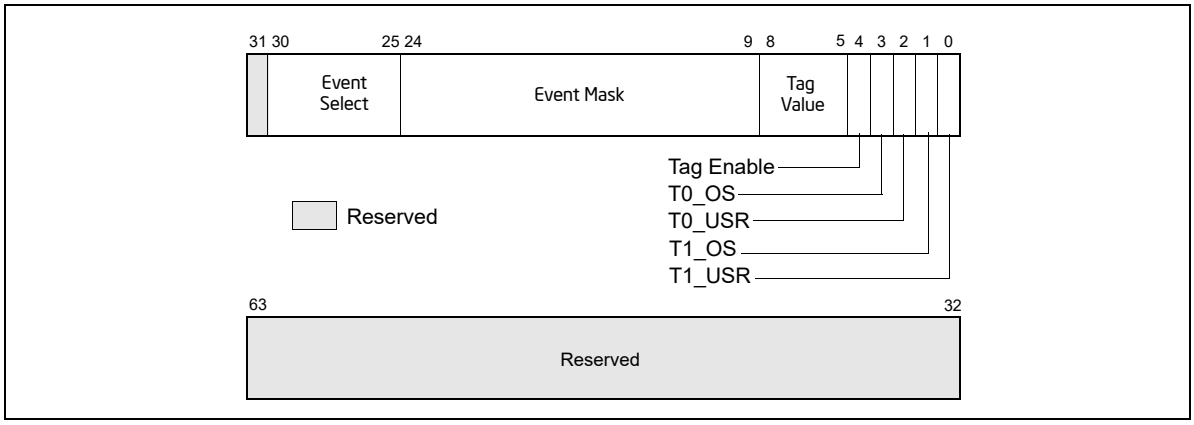


Figure 18-49. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology

- **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 μops 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 μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, "Identifying Logical Processors in an MP System," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.6.4.4, "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.

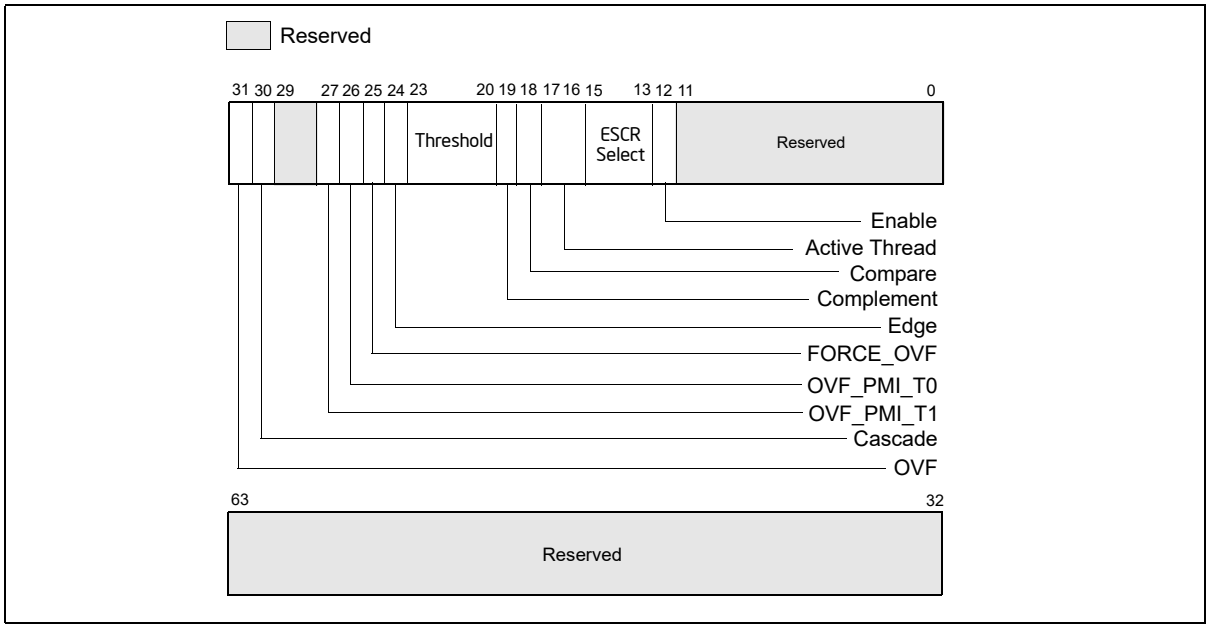


Figure 18-50. Counter Configuration Control Register (CCCR)

- **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 19-31 and 19-32 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 19-37 gives logical processor specific information (TS or TI) for each of the events described in Tables 19-31 and 19-32. If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-73) 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 18-73. 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 18-74.

Table 18-74. 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.

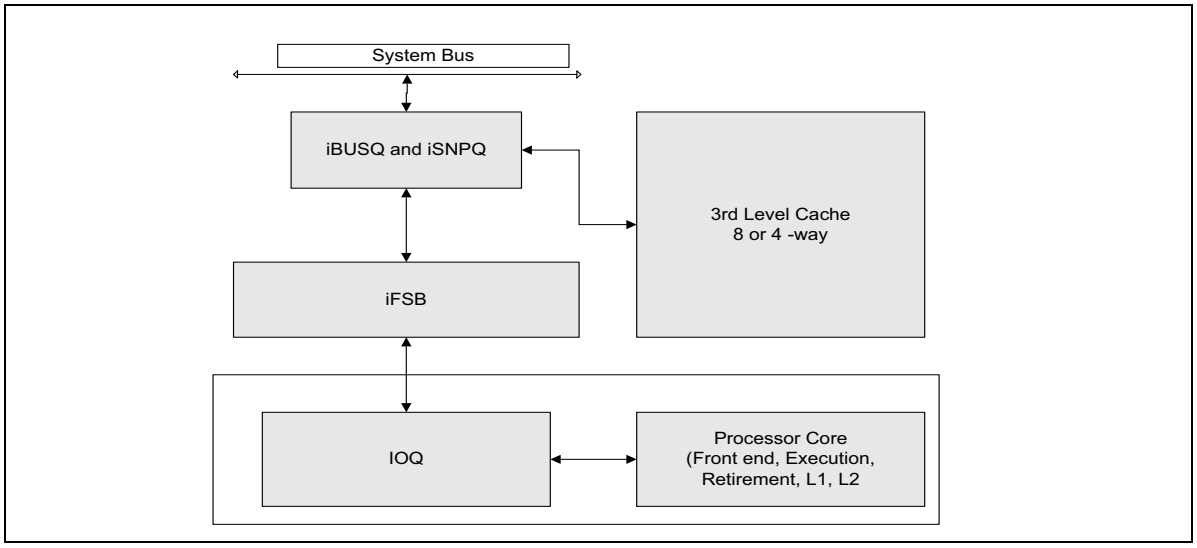


Figure 18-51. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

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.

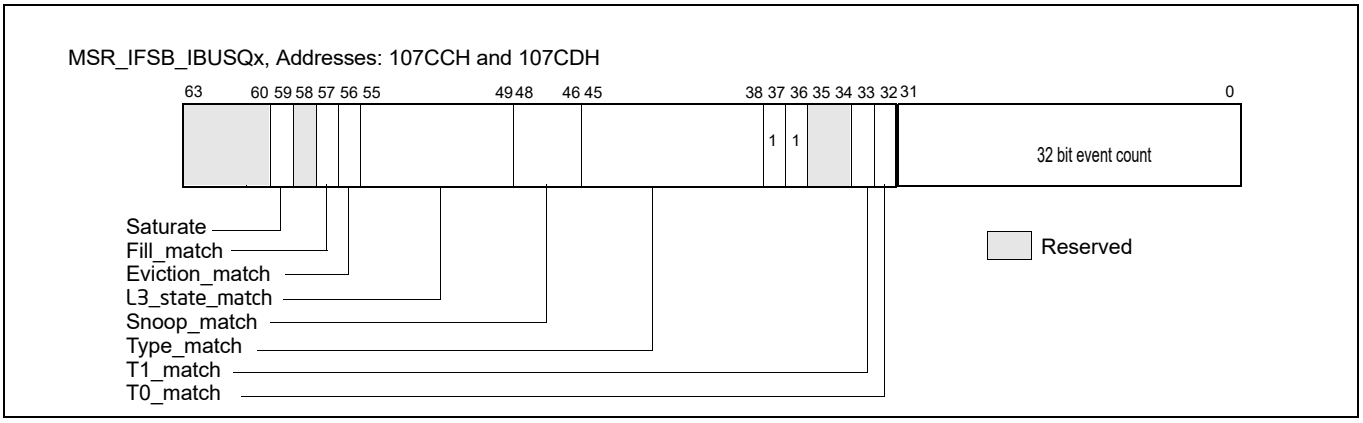


Figure 18-52. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- **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.

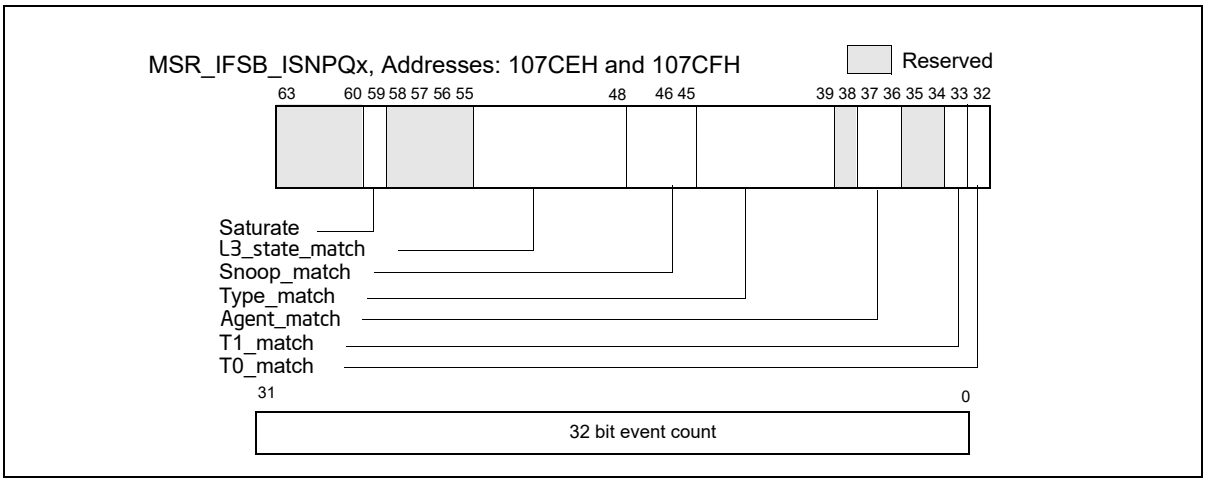


Figure 18-53. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- **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.

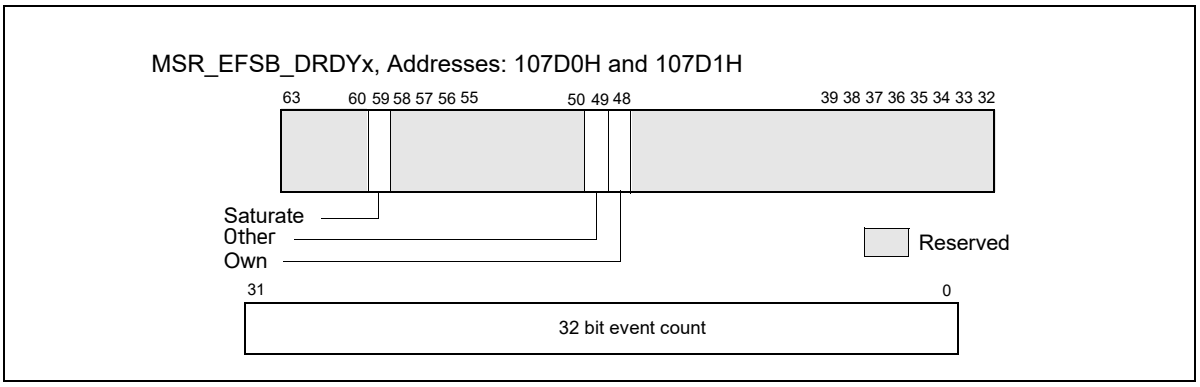


Figure 18-54. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- **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.

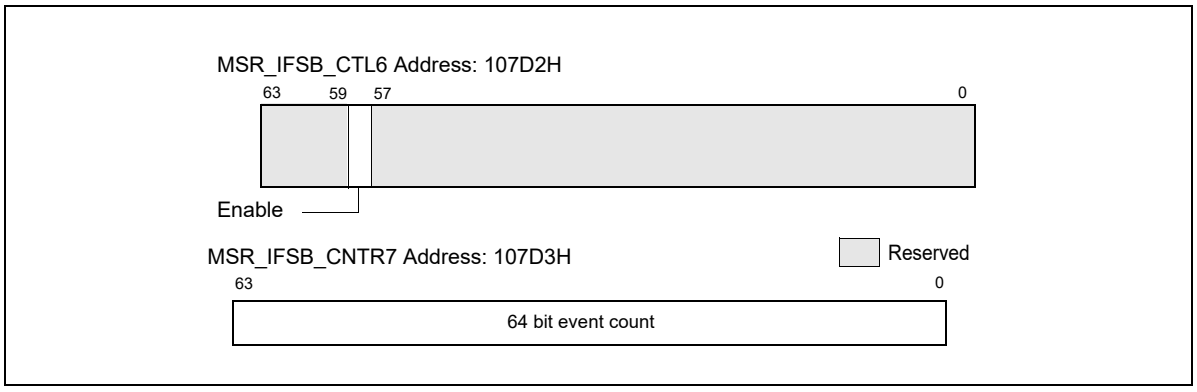


Figure 18-55. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H

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 addi-

tional 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.

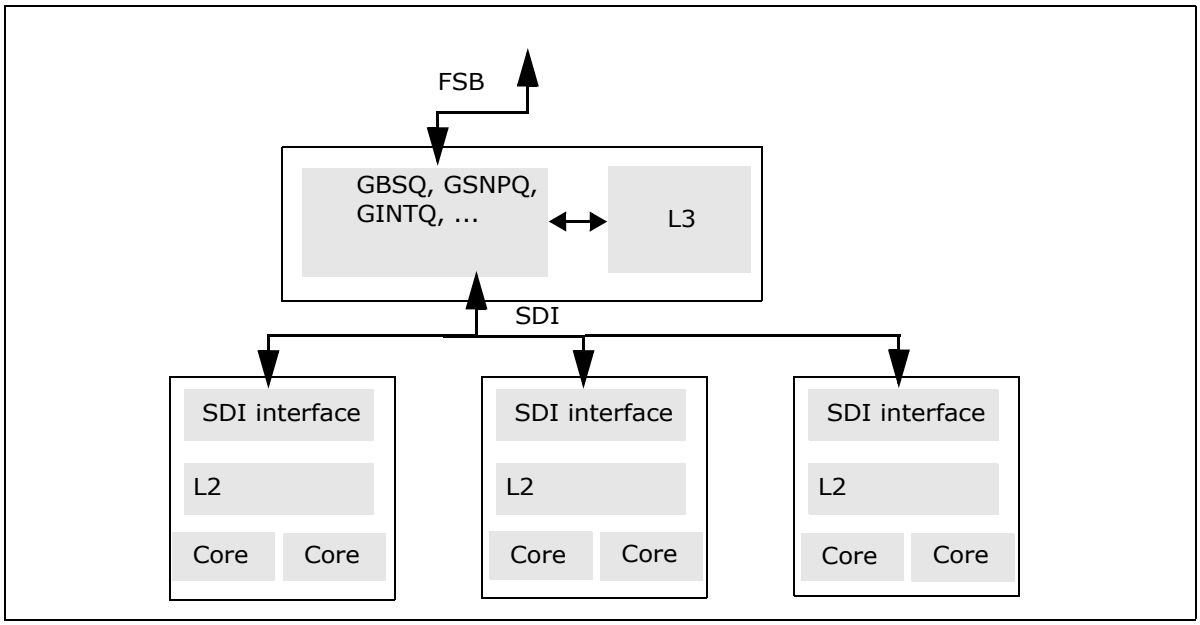


Figure 18-56. Block Diagram of Intel Xeon Processor 7400 Series

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 MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

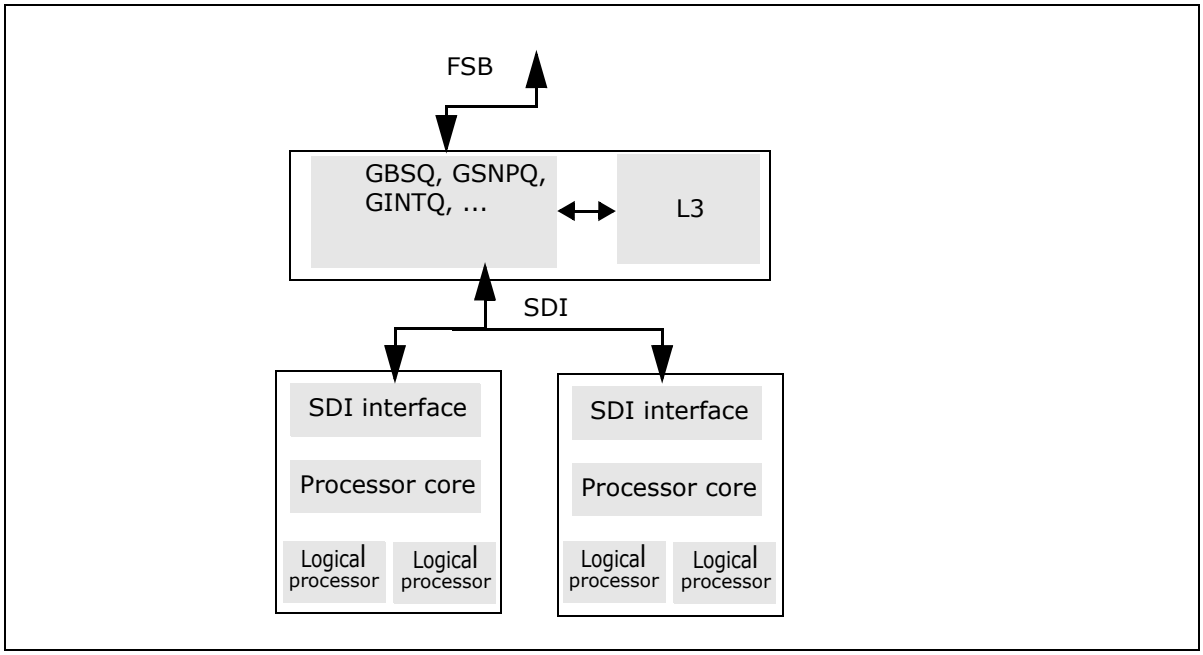


Figure 18-57. Block Diagram of Intel Xeon Processor 7100 Series

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.

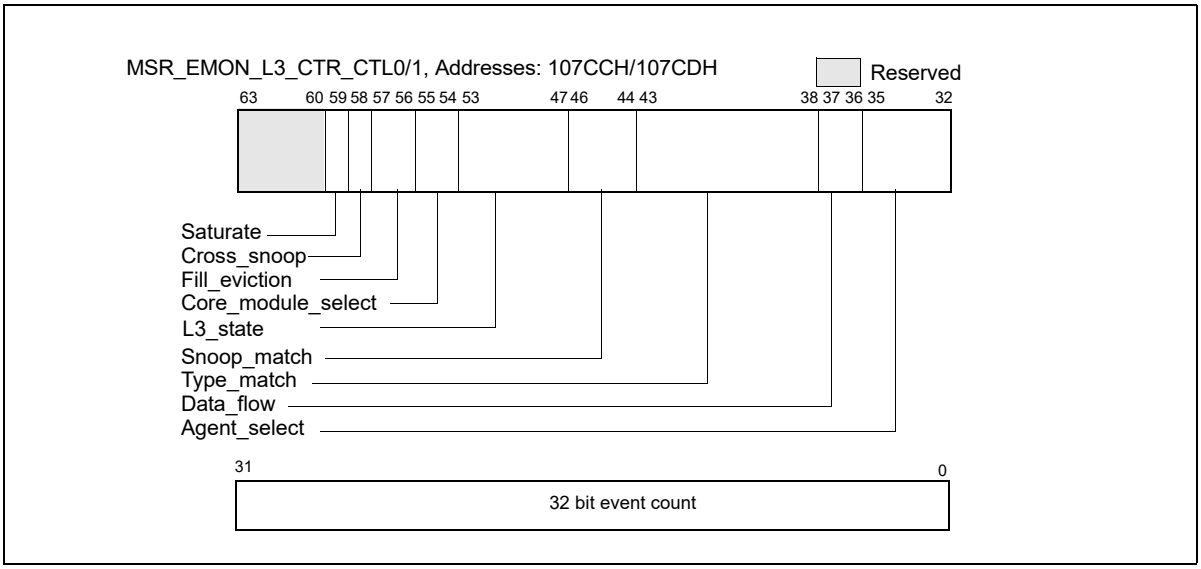


Figure 18-58. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- 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.

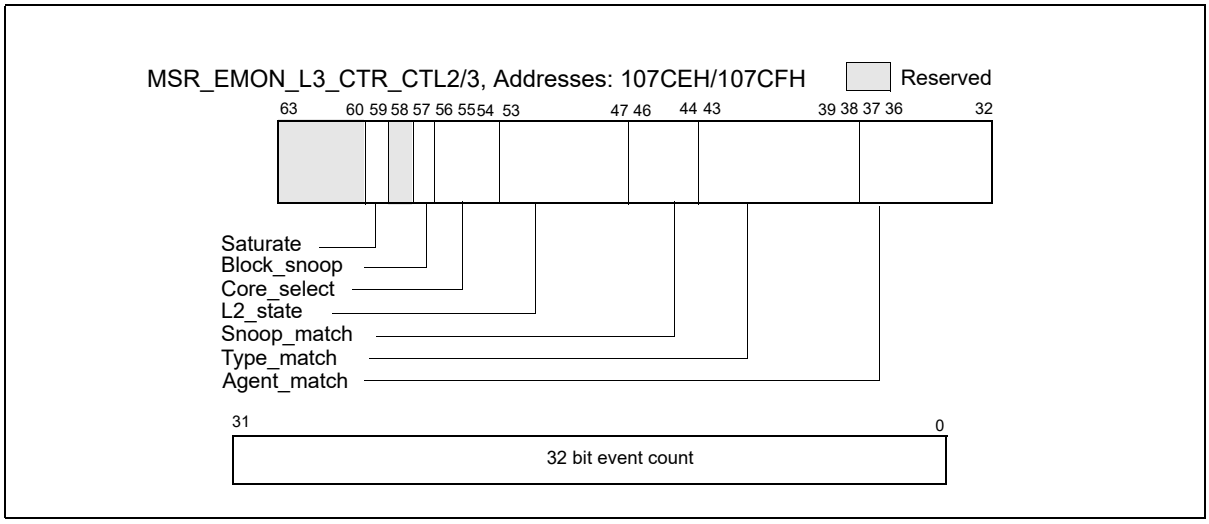


Figure 18-59. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

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.

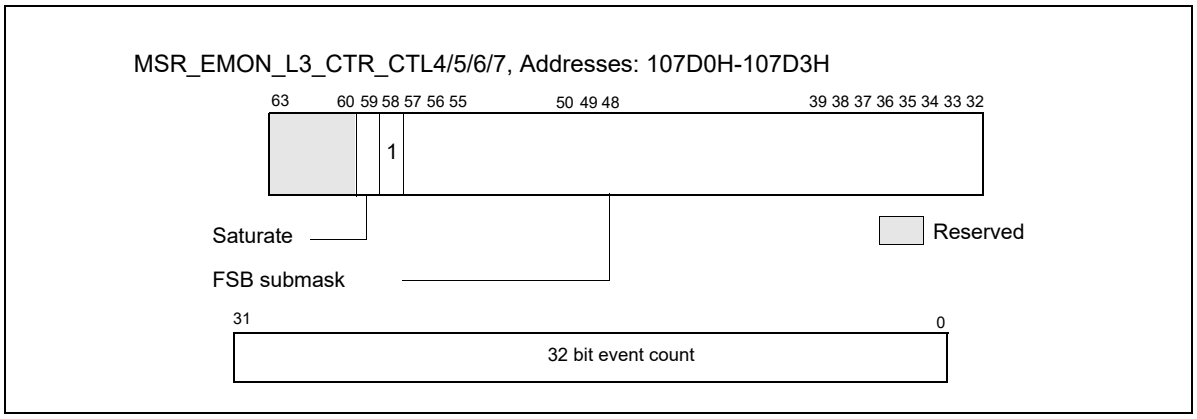


Figure 18-60. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

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 19-40, 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 19-40 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 19-40, 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 19-40).

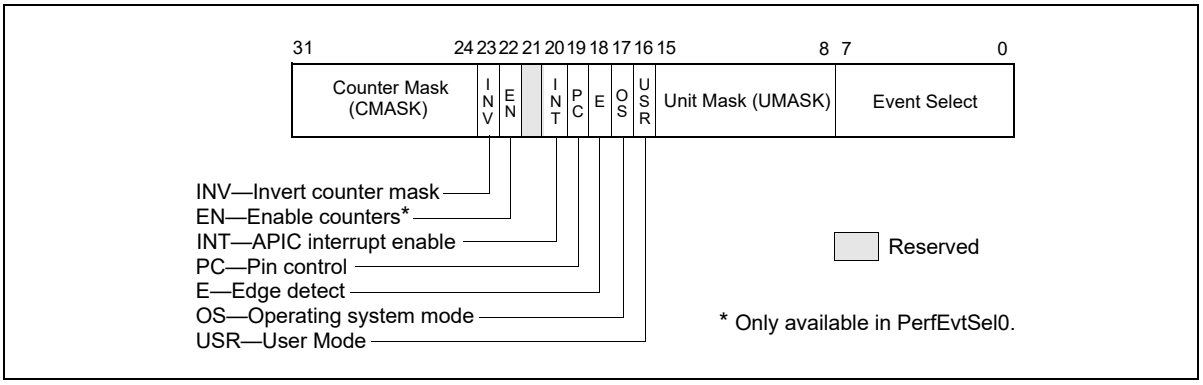


Figure 18-61. PerfEvtSel0 and PerfEvtSel1 MSRs

- **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 PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

18.6.8.2 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 19-41 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 19-41 for a list of available event codes.

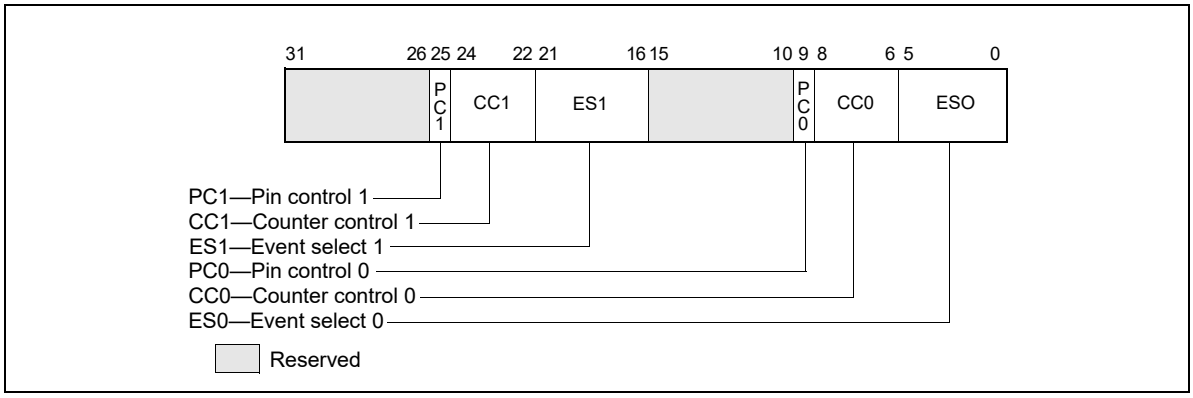


Figure 18-62. CESR MSR (Pentium Processor Only)

- **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 signalling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

18.6.9.2 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 then $2^{40} - 1$. After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

18.6.9.3 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:

Nominal TSC frequency = (CPUID.15H.ECX[31:0] * CPUID.15H.EBX[31:0]) ÷ CPUID.15H.EAX[31:0]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 18-75 can be used to look up the nominal core crystal clock frequency.

Table 18-75. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series[1]	Nominal Core Crystal Clock Frequency
Future Intel® Xeon® processors with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors (does not include Intel® Xeon® processors).	24 MHz
Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

NOTES:

1. For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

18.7.3.1 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 18.2.5.

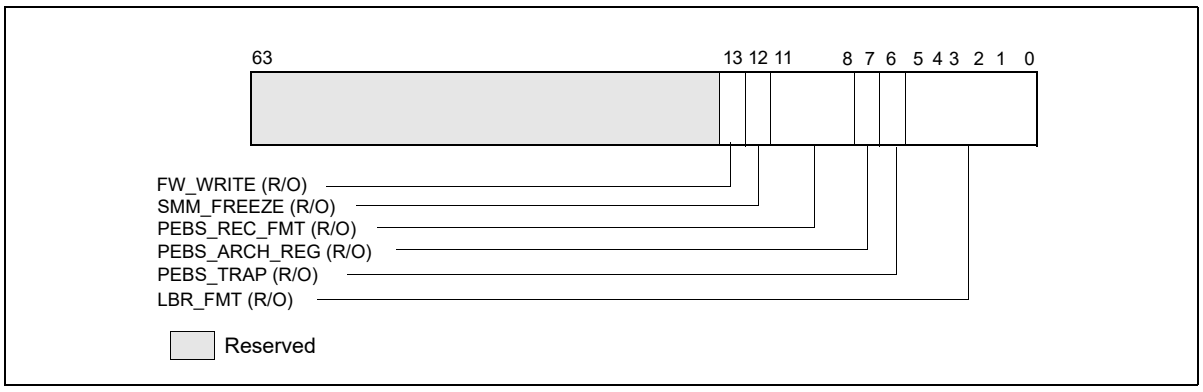


Figure 18-63. Layout of IA32_PERF_CAPABILITIES MSR

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.

19. Updates to Chapter 19, Volume 3B

Change bars show changes to Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Added ROB_MISC_EVENTS.LBR_INSERTS event to Table 19-4 "Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture".

CHAPTER 19
PERFORMANCE MONITORING EVENTS

This chapter lists the performance monitoring events that can be monitored with the Intel 64 or IA-32 processors. The ability to monitor performance events and the events that can be monitored in these processors are mostly model-specific, except for architectural performance events, described in Section 19.1.

Model-specific performance events are listed for each generation of microarchitecture:

- Section 19.2 - Processors based on Skylake microarchitecture
- Section 19.3 - Processors based on Skylake and Kaby Lake microarchitectures
- Section 19.4 - Processors based on Knights Landing microarchitecture
- Section 19.5 - Processors based on Broadwell microarchitecture
- Section 19.6 - Processors based on Haswell microarchitecture
- Section 19.6.1 - Processors based on Haswell-E microarchitecture
- Section 19.7 - Processors based on Ivy Bridge microarchitecture
- Section 19.7.1 - Processors based on Ivy Bridge-E microarchitecture
- Section 19.8 - Processors based on Sandy Bridge microarchitecture
- Section 19.9 - Processors based on Intel® microarchitecture code name Nehalem
- Section 19.10 - Processors based on Intel® microarchitecture code name Westmere
- Section 19.11 - Processors based on Enhanced Intel® Core™ microarchitecture
- Section 19.12 - Processors based on Intel® Core™ microarchitecture
- Section 19.13 - Processors based on the Goldmont microarchitecture
- Section 19.15 - Processors based on the Silvermont microarchitecture
- Section 19.15.1 - Processors based on the Airmont microarchitecture
- Section 19.16 - 45 nm and 32 nm Intel® Atom™ Processors
- Section 19.17 - Intel® Core™ Solo and Intel® Core™ Duo processors
- Section 19.18 - Processors based on Intel NetBurst® microarchitecture
- Section 19.19 - Pentium® M family processors
- Section 19.20 - P6 family processors
- Section 19.21 - Pentium® processors

NOTE

These performance monitoring events are intended to be used as guides for performance tuning. The counter values reported by the performance monitoring events are approximate and believed to be useful as relative guides for tuning software. Known discrepancies are documented where applicable.

All performance event encodings not documented in the appropriate tables for the given processor are considered reserved, and their use will result in undefined counter updates with associated overflow actions.

The event tables listed this chapter provide information for tool developers to support architectural and model-specific performance monitoring events. The tables are up to date at processor launch, but are subject to changes. The most up to date event tables and additional details of performance event implementation for end-user (including additional details beyond event code/umask) can found at the “perfmon” repository provided by The Intel Open Source Technology Center (https://download.01.org/perfmon/).

19.1 ARCHITECTURAL PERFORMANCE MONITORING EVENTS

Architectural performance events are introduced in Intel Core Solo and Intel Core Duo processors. They are also supported on processors based on Intel Core microarchitecture. Table 19-1 lists pre-defined architectural performance events that can be configured using general-purpose performance counters and associated event-select registers.

Table 19-1. Architectural Performance Events

Event Num.	Event Mask Name	Umask Value	Description
3CH	UnHalted Core Cycles	00H	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
3CH	UnHalted Reference Cycles[1]	01H	Counts at a fixed frequency whenever the logical processor is in C0 state (not halted).
C0H	Instructions Retired	00H	Counts when the last uop of an instruction retires.
2EH	LLC Reference	4FH	Counts requests originating from the core that reference a cache line in the last level on-die cache.
2EH	LLC Misses	41H	Counts each cache miss condition for references to the last level on-die cache.
C4H	Branch Instruction Retired	00H	Counts when the last uop of a branch instruction retires.
C5H	Branch Misses Retired	00H	Counts when the last uop of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.

NOTES:

1. Current implementations count at core crystal clock, TSC, or bus clock frequency.

Fixed-function performance counters count only events defined in Table 19-2.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD/CPU_CLK_UNHALTED.CORE/CPU_CLK_UNHALTED.THREAD_ANY	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. If there are more than one logical processor in a processor core, CPU_CLK_UNHALTED.THREAD_ANY is supported by programming IA32_FIXED_CTR_CTRL[bit 6]AnyThread = 1. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events (Contd.)

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

19.2 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR SCALABLE FAMILY

The Intel® Xeon® Processor Scalable Family is based on the Skylake microarchitecture. These processors support the architectural performance monitoring events listed in Table 19-1. Fixed counters in the core PMU support the architecture events defined in Table 19-2. Model-specific performance monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following value: 06_55H .

The comment column in Table 19-4 uses abbreviated letters to indicate additional conditions applicable to the Event Mask Mnemonic. For event umasks listed in Table 19-4 that do not show "AnyT", users should refrain from programming "AnyThread =1" in IA32_PERF_EVTSELx.

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	INST_RETIRED.ANY	Counts the number of instructions retired from execution. For instructions that consist of multiple micro-ops, Counts the retirement of the last micro-op of the instruction. Counting continues during hardware interrupts, traps, and inside interrupt handlers. Notes: INST_RETIRED.ANY is counted by a designated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events. INST_RETIRED.ANY_P is counted by a programmable counter and it is an architectural performance event. Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.	Fixed Counter
00H	02H	CPU_CLK_UNHALTED.THREAD	Counts the number of core cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time. When the core frequency is constant, this event can approximate elapsed time while the core was not in the halt state. It is counted on a dedicated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events.	Fixed Counter

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	02H	CPU_CLK_UNHALTED.THREAD_ANY	Core cycles when at least one thread on the physical core is not in halt state.	AnyThread=1
00H	03H	CPU_CLK_UNHALTED.REF_TSC	Counts the number of reference cycles when the core is not in a halt 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 (for example, P states, TM2 transitions) but has the same incrementing frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state. This event has a constant ratio with the CPU_CLK_UNHALTED.REF_XCLK event. It is counted on a dedicated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events. Note: On all current platforms this event stops counting during 'throttling (TM)' states duty off periods the processor is 'halted'. The counter update is done at a lower clock rate then the core clock the overflow status bit for this counter may appear 'sticky'. After the counter has overflowed and software clears the overflow status bit and resets the counter to less than MAX. The reset value to the counter is not clocked immediately so the overflow status bit will flip "high (1)" and generate another PMI (if enabled) after which the reset value gets clocked into the counter. Therefore, software will get the interrupt, read the overflow status bit '1 for bit 34 while the counter value is less than MAX. Software should ignore this case.	Fixed Counter
03H	02H	LD_BLOCKS.STORE_FORWARD	Counts how many times the load operation got the true Block-on-Store blocking code preventing store forwarding. This includes cases when: a. preceding store conflicts with the load (incomplete overlap), b. store forwarding is impossible due to u-arch limitations, c. preceding lock RMW operations are not forwarded, d. store has the no-forward bit set (uncacheable/page-split/masked stores), e. all-blocking stores are used (mostly, fences and port I/O), and others. The most common case is a load blocked due to its address range overlapping with a preceding smaller uncompleted store. Note: This event does not take into account cases of out-of-SW-control (for example, SbTailHit), unknown physical STA, and cases of blocking loads on store due to being non-WB memory type or a lock. These cases are covered by other events. See the table of not supported store forwards in the Optimization Guide.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	Counts false dependencies in MOB when the partial comparison upon loose net check and dependency was resolved by the Enhanced Loose net mechanism. This may not result in high performance penalties. Loose net checks can fail when loads and stores are 4k aliased.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Counts demand data loads that caused a page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels, but the walk need not have completed.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Counts demand data loads that caused a completed page walk (4K page size). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Counts demand data loads that caused a completed page walk (2M and 4M page sizes). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	08H	DTLB_LOAD_MISSES.WALK_COMPLETED_1G	Counts load misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts demand data loads that caused a completed page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a load. EPT page walk duration are excluded in Skylake microarchitecture.	
08H	10H	DTLB_LOAD_MISSES.WALK_ACTIVE	Counts cycles when at least one PMH (Page Miss Handler) is busy with a page walk for a load.	CounterMask=1 CMSK1
08H	20H	DTLB_LOAD_MISSES.STLB_HIT	Counts loads that miss the DTLB (Data TLB) and hit the STLB (Second level TLB).	
0DH	01H	INT_MISC.RECOVERY_CYCLES	Core cycles the Resource allocator was stalled due to recovery from an earlier branch misprediction or machine clear event.	
0DH	01H	INT_MISC.RECOVERY_CYCLES_ANY	Core cycles the allocator was stalled due to recovery from earlier clear event for any thread running on the physical core (e.g. misprediction or memory nuke).	AnyThread=1 AnyT
0DH	80H	INT_MISC.CLEAR_RESTEER_CYCLES	Cycles the issue-stage is waiting for front-end to fetch from resteered path following branch misprediction or machine clear events.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of uops that the Resource Allocation Table (RAT) issues to the Reservation Station (RS).	
0EH	01H	UOPS_ISSUED.STALL_CYCLES	Counts cycles during which the Resource Allocation Table (RAT) does not issue any uops to the reservation station (RS) for the current thread.	CounterMask=1 Invert=1 CMSK1, INV

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	02H	UOPS_ISSUED.VECTOR_WIDTH_MISMATCH	Counts the number of Blend Uops issued by the Resource Allocation Table (RAT) to the reservation station (RS) in order to preserve upper bits of vector registers. Starting with the Skylake microarchitecture, these Blend uops are needed since every Intel SSE instruction executed in Dirty Upper State needs to preserve bits 128-255 of the destination register. For more information, refer to Mixing Intel AVX and Intel SSE Code section of the Optimization Guide.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA uops being allocated. A uop is generally considered SlowLea if it has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.	
14H	01H	ARITH.DIVIDER_ACTIVE	Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations.	CounterMask=1
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Counts the number of demand Data Read requests that miss L2 cache. Only not rejected loads are counted.	
24H	22H	L2_RQSTS.RFO_MISS	Counts the RFO (Read-for-Ownership) requests that miss L2 cache.	
24H	24H	L2_RQSTS.CODE_RD_MISS	Counts L2 cache misses when fetching instructions.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that miss L2 cache.	
24H	38H	L2_RQSTS.PF_MISS	Counts requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that miss L2 cache.	
24H	3FH	L2_RQSTS.MISS	All requests that miss L2 cache.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Counts the number of demand Data Read requests that hit L2 cache. Only non rejected loads are counted.	
24H	42H	L2_RQSTS.RFO_HIT	Counts the RFO (Read-for-Ownership) requests that hit L2 cache.	
24H	44H	L2_RQSTS.CODE_RD_HIT	Counts L2 cache hits when fetching instructions, code reads.	
24H	D8H	L2_RQSTS.PF_HIT	Counts requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that hit L2 cache.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts the number of demand Data Read requests (including requests from L1D hardware prefetchers). These loads may hit or miss L2 cache. Only non rejected loads are counted.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts the total number of RFO (read for ownership) requests to L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts the total number of L2 code requests.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	Demand requests to L2 cache.	
24H	F8H	L2_RQSTS.ALL_PF	Counts the total number of requests from the L2 hardware prefetchers.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	FFH	L2_RQSTS.REFERENCES	All L2 requests.	
28H	07H	CORE_POWER.LVL0_TURBO_LICENSE	Core cycles where the core was running with power-delivery for baseline license level 0. This includes non-AVX codes, SSE, AVX 128-bit, and low-current AVX 256-bit codes.	
28H	18H	CORE_POWER.LVL1_TURBO_LICENSE	Core cycles where the core was running with power-delivery for license level 1. This includes high current AVX 256-bit instructions as well as low current AVX 512-bit instructions.	
28H	20H	CORE_POWER.LVL2_TURBO_LICENSE	Core cycles where the core was running with power-delivery for license level 2 (introduced in Skylake Server microarchitecture). This includes high current AVX 512-bit instructions.	
28H	40H	CORE_POWER.THROTTLE	Core cycles the out-of-order engine was throttled due to a pending power level request.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts core-originated cacheable requests that miss the L3 cache (Longest Latency cache). Requests include data and code reads, Reads-for-Ownership (RFOs), speculative accesses and hardware prefetches from L1 and L2. It does not include all misses to the L3.	See Table 19-1.
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts core-originated cacheable requests to the L3 cache (Longest Latency cache). Requests include data and code reads, Reads-for-Ownership (RFOs), speculative accesses and hardware prefetches from L1 and L2. It does not include all accesses to the L3.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	This is an architectural event that counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling. For this reason, this event may have a changing ratio with regards to wall clock time.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P_ANY	Core cycles when at least one thread on the physical core is not in halt state.	AnyThread=1 AnyT
3CH	00H	CPU_CLK_UNHALTED.RING0_TRANS	Counts when the Current Privilege Level (CPL) transitions from ring 1, 2 or 3 to ring 0 (Kernel).	EdgeDetect=1 CounterMask=1
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyThread=1 AnyT
3CH	01H	CPU_CLK_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyThread=1 AnyT
3CH	02H	CPU_CLK_THREAD_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	
3CH	02H	CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
48H	01H	L1D_PEND_MISS.PENDING	Counts duration of L1D miss outstanding, that is each cycle number of Fill Buffers (FB) outstanding required by Demand Reads. FB either is held by demand loads, or it is held by non-demand loads and gets hit at least once by demand. The valid outstanding interval is defined until the FB deallocation by one of the following ways: from FB allocation, if FB is allocated by demand from the demand Hit FB, if it is allocated by hardware or software prefetch.Note: In the L1D, a Demand Read contains cacheable or noncacheable demand loads, including ones causing cache-line splits and reads due to page walks resulted from any request type.	
48H	01H	L1D_PEND_MISS.PENDING_CYCLES	Counts duration of L1D miss outstanding in cycles.	CounterMask=1 CMSK1
48H	01H	L1D_PEND_MISS.PENDING_CYCLES_ANY	Cycles with L1D load Misses outstanding from any thread on physical core.	CounterMask=1 AnyThread=1 CMSK1, AnyT
48H	02H	L1D_PEND_MISS.FB_FULL	Number of times a request needed a FB (Fill Buffer) entry but there was no entry available for it. A request includes cacheable/uncacheable demands that are load, store or SW prefetch instructions.	
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Counts demand data stores that caused a page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels, but the walk need not have completed.	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Counts demand data stores that caused a completed page walk (4K page size). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Counts demand data stores that caused a completed page walk (2M and 4M page sizes). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	08H	DTLB_STORE_MISSES.WALK_COMPLETED_1G	Counts store misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Counts demand data stores that caused a completed page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	10H	DTLB_STORE_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a store. EPT page walk duration are excluded in Skylake microarchitecture.	
49H	10H	DTLB_STORE_MISSES.WALK_ACTIVE	Counts cycles when at least one PMH (Page Miss Handler) is busy with a page walk for a store.	CounterMask=1 CMSK1
49H	20H	DTLB_STORE_MISSES.STLB_HIT	Stores that miss the DTLB (Data TLB) and hit the STLB (2nd Level TLB).	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4CH	01H	LOAD_HIT_PRE.SW_PF	Counts all not software-prefetch load dispatches that hit the fill buffer (FB) allocated for the software prefetch. It can also be incremented by some lock instructions. So it should only be used with profiling so that the locks can be excluded by ASM (Assembly File) inspection of the nearby instructions.	
4FH	10H	EPT.WALK_PENDING	Counts cycles for each PMH (Page Miss Handler) that is busy with an EPT (Extended Page Table) walk for any request type.	
51H	01H	L1D.REPLACEMENT	Counts L1D data line replacements including opportunistic replacements, and replacements that require stall-for-replace or block-for-replace.	
54H	01H	TX_MEM.ABORT_CONFLICT	Number of times a TSX line had a cache conflict.	
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	
54H	04H	TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a TSX Abort was triggered due to a non-release/commit store to lock.	
54H	08H	TX_MEM.ABORT_HLE_ELISION_BUFFER_NOT_EMPTY	Number of times a TSX Abort was triggered due to commit but Lock Buffer not empty.	
54H	10H	TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH	Number of times a TSX Abort was triggered due to release/commit but data and address mismatch.	
54H	20H	TX_MEM.ABORT_HLE_ELISION_BUFFER_UNSUPPORTED_ALIGNMENT	Number of times a TSX Abort was triggered due to attempting an unsupported alignment from Lock Buffer.	
54H	40H	TX_MEM.HLE_ELISION_BUFFER_FULL	Number of times we could not allocate Lock Buffer.	
5DH	01H	TX_EXEC.MISC1	Unfriendly TSX abort triggered by a flowmarker.	
5DH	02H	TX_EXEC.MISC2	Unfriendly TSX abort triggered by a vzeroupper instruction.	
5DH	04H	TX_EXEC.MISC3	Unfriendly TSX abort triggered by a nest count that is too deep.	
5DH	08H	TX_EXEC.MISC4	RTM region detected inside HLE.	
5DH	10H	TX_EXEC.MISC5	Counts the number of times an HLE XACQUIRE instruction was executed inside an RTM transactional region.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Counts cycles during which the reservation station (RS) is empty for the thread.; Note: In ST-mode, not active thread should drive 0. This is usually caused by severely costly branch mispredictions, or allocator/FE issues.	
5EH	01H	RS_EVENTS.EMPTY_END	Counts end of periods where the Reservation Station (RS) was empty. Could be useful to precisely locate front-end Latency Bound issues.	EdgeDetect=1 CounterMask=1 Invert=1 CMSK1, INV

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Counts the number of offcore outstanding Demand Data Read transactions in the super queue (SQ) every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor. See the corresponding Umask under OFFCORE_REQUESTS. Note: A prefetch promoted to Demand is counted from the promotion point.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_DATA_RD	Counts cycles when offcore outstanding Demand Data Read transactions are present in the super queue (SQ). A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation).	CounterMask=1 CMSK1
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD_GE_6	Cycles with at least 6 offcore outstanding Demand Data Read transactions in uncore queue.	CounterMask=6 CMSK6
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Counts the number of offcore outstanding Code Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_CODE_RD	Counts the number of offcore outstanding Code Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Counts the number of offcore outstanding RFO (store) transactions in the super queue (SQ) every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO	Counts the number of offcore outstanding demand rfo Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CMSK1
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Counts the number of offcore outstanding cacheable Core Data Read transactions in the super queue every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DATA_RD	Counts cycles when offcore outstanding cacheable Core Data Read transactions are present in the super queue. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD	Counts number of Offcore outstanding Demand Data Read requests that miss L3 cache in the superQ every cycle.	
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_L3_MISS_DEMAND_DATA_RD	Cycles with at least 1 Demand Data Read requests who miss L3 cache in the superQ.	CounterMask=1 CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD_GE_6	Cycles with at least 6 Demand Data Read requests that miss L3 cache in the superQ.	CounterMask=6 CMSK6
79H	04H	IDQ.MITE_UOPS	Counts the number of uops delivered to Instruction Decode Queue (IDQ) from the MITE path. Counting includes uops that may 'bypass' the IDQ. This also means that uops are not being delivered from the Decode Stream Buffer (DSB).	
79H	04H	IDQ.MITE_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) from the MITE path. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	08H	IDQ.DSB_UOPS	Counts the number of uops delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Counting includes uops that may 'bypass' the IDQ.	
79H	08H	IDQ.DSB_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	10H	IDQ.MS_DSB_CYCLES	Counts cycles during which uops initiated by Decode Stream Buffer (DSB) are being delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts the number of cycles 4 uops were delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Count includes uops that may 'bypass' the IDQ.	CounterMask=4 CMSK4
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts the number of cycles uops were delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Count includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	20H	IDQ.MS_MITE_UOPS	Counts the number of uops initiated by MITE and delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts the number of cycles 4 uops were delivered to the Instruction Decode Queue (IDQ) from the MITE (legacy decode pipeline) path. Counting includes uops that may 'bypass' the IDQ. During these cycles uops are not being delivered from the Decode Stream Buffer (DSB).	CounterMask=4 CMSK4

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts the number of cycles uops were delivered to the Instruction Decode Queue (IDQ) from the MITE (legacy decode pipeline) path. Counting includes uops that may 'bypass' the IDQ. During these cycles uops are not being delivered from the Decode Stream Buffer (DSB).	CounterMask=1 CMSK1
79H	30H	IDQ.MS_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ. Uops maybe initiated by Decode Stream Buffer (DSB) or MITE.	CounterMask=1 CMSK1
79H	30H	IDQ.MS_SWITCHES	Number of switches from DSB (Decode Stream Buffer) or MITE (legacy decode pipeline) to the Microcode Sequencer.	EdgeDetect=1 CounterMask=1 EDGE
79H	30H	IDQ.MS_UOPS	Counts the total number of uops delivered by the Microcode Sequencer (MS). Any instruction over 4 uops will be delivered by the MS. Some instructions such as transcendentals may additionally generate uops from the MS.	
80H	04H	ICACHE_16B.IFDATA_STALL	Cycles where a code line fetch is stalled due to an L1 instruction cache miss. The legacy decode pipeline works at a 16 Byte granularity.	
83H	01H	ICACHE_64B.IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	02H	ICACHE_64B.IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	04H	ICACHE_64B.IFTAG_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Counts page walks of any page size (4K/2M/4M/1G) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB, but the walk need not have completed.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Counts completed page walks (4K page size) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Counts completed page walks of any page size (4K/2M/4M/1G) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	
85H	08H	ITLB_MISSES.WALK_COMPLETED_1G	Counts store misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Counts completed page walks (2M and 4M page sizes) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	10H	ITLB_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for an instruction fetch request. EPT page walk duration are excluded in Skylake microarchitecture.	
85H	10H	ITLB_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a page walk for code (instruction fetch) request. EPT page walk duration are excluded in Skylake microarchitecture.	CounterMask=1
85H	20H	ITLB_MISSES.STLB_HIT	Instruction fetch requests that miss the ITLB and hit the STLB.	
87H	01H	ILD_STALL.LCP	Counts cycles that the Instruction Length decoder (ILD) stalls occurred due to dynamically changing prefix length of the decoded instruction (by operand size prefix instruction 0x66, address size prefix instruction 0x67 or REX.W for Intel64). Count is proportional to the number of prefixes in a 16B-line. This may result in a three-cycle penalty for each LCP (Length changing prefix) in a 16-byte chunk.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Counts the number of uops not delivered to Resource Allocation Table (RAT) per thread adding “4 – x” ? when Resource Allocation Table (RAT) is not stalled and Instruction Decode Queue (IDQ) delivers x uops to Resource Allocation Table (RAT) (where x belongs to {0,1,2,3}). Counting does not cover cases when: a. IDQ-Resource Allocation Table (RAT) pipe serves the other thread. b. Resource Allocation Table (RAT) is stalled for the thread (including uop drops and clear BE conditions). c. Instruction Decode Queue (IDQ) delivers four uops.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE	Counts, on the per-thread basis, cycles when no uops are delivered to Resource Allocation Table (RAT). IDQ_Uops_Not_Delivered.core =4.	CounterMask=4 CMSK4
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE	Counts, on the per-thread basis, cycles when less than 1 uop is delivered to Resource Allocation Table (RAT). IDQ_Uops_Not_Delivered.core >= 3.	CounterMask=3 CMSK3
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE	Cycles with less than 2 uops delivered by the front end.	CounterMask=2 CMSK2
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE	Cycles with less than 3 uops delivered by the front end.	CounterMask=1 CMSK1
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	Counts cycles FE delivered 4 uops or Resource Allocation Table (RAT) was stalling FE.	CounterMask=1 Invert=1 CMSK, INV
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 2.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 3.	
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 4.	
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 5.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 6.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 7.	
A2H	01H	RESOURCE_STALLS.ANY	Counts resource-related stall cycles. Reasons for stalls can be as follows: a. *any* u-arch structure got full (LB, SB, RS, ROB, BOB, LM, Physical Register Reclaim Table (PRRT), or Physical History Table (PHT) slots). b. *any* u-arch structure got empty (like INT/SIMD FreeLists). c. FPU control word (FPCW), MXCSR.and others. This counts cycles that the pipeline back end blocked uop delivery from the front end.	
A2H	08H	RESOURCE_STALLS.SB	Counts allocation stall cycles caused by the store buffer (SB) being full. This counts cycles that the pipeline back end blocked uop delivery from the front end.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_MISS	Cycles while L2 cache miss demand load is outstanding.	CounterMask=1 CMSK1
A3H	02H	CYCLE_ACTIVITY.CYCLES_L3_MISS	Cycles while L3 cache miss demand load is outstanding.	CounterMask=2 CMSK2
A3H	04H	CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls.	CounterMask=4 CMSK4
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_MISS	Execution stalls while L2 cache miss demand load is outstanding.	CounterMask=5 CMSK5
A3H	06H	CYCLE_ACTIVITY.STALLS_L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	CounterMask=6 CMSK6
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_MISS	Cycles while L1 cache miss demand load is outstanding.	CounterMask=8 CMSK8
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_MISS	Execution stalls while L1 cache miss demand load is outstanding.	CounterMask=12 CMSK12
A3H	10H	CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load.	CounterMask=16 CMSK16
A3H	14H	CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load.	CounterMask=20 CMSK20
A6H	01H	EXE_ACTIVITY.EXE_BOUND_0_PORTS	Counts cycles during which no uops were executed on all ports and Reservation Station (RS) was not empty.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A6H	02H	EXE_ACTIVITY.1_PORTS_UTIL	Counts cycles during which a total of 1 uop was executed on all ports and Reservation Station (RS) was not empty.	
A6H	04H	EXE_ACTIVITY.2_PORTS_UTIL	Counts cycles during which a total of 2 uops were executed on all ports and Reservation Station (RS) was not empty.	
A6H	08H	EXE_ACTIVITY.3_PORTS_UTIL	Cycles total of 3 uops are executed on all ports and Reservation Station (RS) was not empty.	
A6H	10H	EXE_ACTIVITY.4_PORTS_UTIL	Cycles total of 4 uops are executed on all ports and Reservation Station (RS) was not empty.	
A6H	40H	EXE_ACTIVITY.BOUND_ON_STORES	Cycles where the Store Buffer was full and no outstanding load.	
A8H	01H	LSD.UOPS	Number of uops delivered to the back-end by the LSD (Loop Stream Detector).	
A8H	01H	LSD.CYCLES_ACTIVE	Counts the cycles when at least one uop is delivered by the LSD (Loop-stream detector).	CounterMask=1 CMSK1
A8H	01H	LSD.CYCLES_4_UOPS	Counts the cycles when 4 uops are delivered by the LSD (Loop-stream detector).	CounterMask=4 CMSK4
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Counts Decode Stream Buffer (DSB)-to-MITE switch true penalty cycles. These cycles do not include uops routed through because of the switch itself, for example, when Instruction Decode Queue (IDQ) pre-allocation is unavailable, or Instruction Decode Queue (IDQ) is full. SBD-to-MITE switch true penalty cycles happen after the merge mux (MM) receives Decode Stream Buffer (DSB) Sync-indication until receiving the first MITE uop. MM is placed before Instruction Decode Queue (IDQ) to merge uops being fed from the MITE and Decode Stream Buffer (DSB) paths. Decode Stream Buffer (DSB) inserts the Sync-indication whenever a Decode Stream Buffer (DSB)-to-MITE switch occurs.Penalty: A Decode Stream Buffer (DSB) hit followed by a Decode Stream Buffer (DSB) miss can cost up to six cycles in which no uops are delivered to the IDQ. Most often, such switches from the Decode Stream Buffer (DSB) to the legacy pipeline cost 0 to 2 cycles.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of flushes of the big or small ITLB pages. Counting include both TLB Flush (covering all sets) and TLB Set Clear (set-specific).	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Counts the Demand Data Read requests sent to uncore. Use it in conjunction with OFFCORE_REQUESTS_OUTSTANDING to determine average latency in the uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Counts both cacheable and non-cacheable code read requests.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Counts the demand RFO (read for ownership) requests including regular RFOs, locks, ItoM.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Counts the demand and prefetch data reads. All Core Data Reads include cacheable 'Demands' and L2 prefetchers (not L3 prefetchers). Counting also covers reads due to page walks resulted from any request type.	
B0H	10H	OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD	Demand Data Read requests who miss L3 cache.	
B0H	80H	OFFCORE_REQUESTS.ALL_REQUESTS	Counts memory transactions reached the super queue including requests initiated by the core, all L3 prefetches, page walks, etc.	
B1H	01H	UOPS_EXECUTED.THREAD	Number of uops to be executed per-thread each cycle.	
B1H	01H	UOPS_EXECUTED.STALL_CYCLES	Counts cycles during which no uops were dispatched from the Reservation Station (RS) per thread.	CounterMask=1 Invert=1 CMSK, INV
B1H	01H	UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC	Cycles where at least 1 uop was executed per-thread.	CounterMask=1 CMSK1
B1H	01H	UOPS_EXECUTED.CYCLES_GE_2_UOPS_EXEC	Cycles where at least 2 uops were executed per-thread.	CounterMask=2 CMSK2
B1H	01H	UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC	Cycles where at least 3 uops were executed per-thread.	CounterMask=3 CMSK3
B1H	01H	UOPS_EXECUTED.CYCLES_GE_4_UOPS_EXEC	Cycles where at least 4 uops were executed per-thread.	CounterMask=4 CMSK4
B1H	02H	UOPS_EXECUTED.CORE	Number of uops executed from any thread.	
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_1	Cycles at least 1 micro-op is executed from any thread on physical core.	CounterMask=1 CMSK1
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_2	Cycles at least 2 micro-op is executed from any thread on physical core.	CounterMask=2 CMSK2
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_3	Cycles at least 3 micro-op is executed from any thread on physical core.	CounterMask=3 CMSK3
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_4	Cycles at least 4 micro-op is executed from any thread on physical core.	CounterMask=4 CMSK4
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_NONE	Cycles with no micro-ops executed from any thread on physical core.	CounterMask=1 Invert=1 CMSK1, INV
B1H	10H	UOPS_EXECUTED.X87	Counts the number of x87 uops executed.	
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Counts the number of cases when the offcore requests buffer cannot take more entries for the core. This can happen when the superqueue does not contain eligible entries, or when L1D writeback pending FIFO requests is full. Note: Writeback pending FIFO has six entries.	
BDH	01H	TLB_FLUSH.DTLB_THREAD	Counts the number of DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Counts the number of any STLB flush attempts (such as entire, VPID, PCID, InvPage, CR3 write, etc.).	
C0H	00H	INST_RETIRED.ANY_P	Counts the number of instructions (EOMs) retired. Counting covers macro-fused instructions individually (that is, increments by two).	See Table 19-1.

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	01H	INST_RETIRED.PREC_DIST	A version of INST_RETIRED that allows for a more unbiased distribution of samples across instructions retired. It utilizes the Precise Distribution of Instructions Retired (PDIR) feature to mitigate some bias in how retired instructions get sampled.	Precise event capable Requires PEBS on General Counter 1(PDIR).
C1H	3FH	OTHER_ASSISTS.ANY	Number of times a microcode assist is invoked by HW other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists.	
C2H	01H	UOPS_RETIRED.STALL_CYCLES	This is a non-precise version (that is, does not use PEBS) of the event that counts cycles without actually retired uops.	CounterMask=1 Invert=1 CMSK1, INV
C2H	01H	UOPS_RETIRED.TOTAL_CYCLES	Number of cycles using always true condition (uops_ret < 16) applied to non PEBS uops retired event.	CounterMask=10 Invert=1 CMSK10, INV
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the retirement slots used.	
C3H	01H	MACHINE_CLEARS.COUNT	Number of machine clears (nukes) of any type.	EdgeDetect=1 CounterMask=1 CMSK1, EDG
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of memory ordering Machine Clears detected. Memory Ordering Machine Clears can result from one of the following: a. memory disambiguation, b. external snoop, or c. cross SMT-HW-thread snoop (stores) hitting load buffer.	
C3H	04H	MACHINE_CLEARS.SMC	Counts self-modifying code (SMC) detected, which causes a machine clear.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts all (macro) branch instructions retired.	Precise event capable. See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	This is a non-precise version (that is, does not use PEBS) of the event that counts conditional branch instructions retired.	Precise event capable. PS
C4H	02H	BR_INST_RETIRED.NEAR_CALL	This is a non-precise version (that is, does not use PEBS) of the event that counts both direct and indirect near call instructions retired.	Precise event capable. PS
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	This is a non-precise version (that is, does not use PEBS) of the event that counts return instructions retired.	Precise event capable. PS
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	This is a non-precise version (that is, does not use PEBS) of the event that counts not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	This is a non-precise version (that is, does not use PEBS) of the event that counts taken branch instructions retired.	Precise event capable. PS
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	This is a non-precise version (that is, does not use PEBS) of the event that counts far branch instructions retired.	Precise event capable. PS

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts all the retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor incorrectly predicts the destination of the branch. When the misprediction is discovered at execution, all the instructions executed in the wrong (speculative) path must be discarded, and the processor must start fetching from the correct path.	Precise event capable. See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	This is a non-precise version (that is, does not use PEBS) of the event that counts mispredicted conditional branch instructions retired.	Precise event capable. PS
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts both taken and not taken retired mispredicted direct and indirect near calls, including both register and memory indirect.	Precise event capable.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were mispredicted and taken.	Precise event capable. PS
C6H	01H	FRONTEND_RETIRED.DSB_MISS	Counts retired Instructions that experienced DSB (Decode stream buffer, i.e. the decoded instruction-cache) miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.L1I_MISS	Retired Instructions who experienced Instruction L1 Cache true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.L2_MISS	Retired Instructions who experienced Instruction L2 Cache true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.ITLB_MISS	Counts retired Instructions that experienced iTLB (Instruction TLB) true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.STLB_MISS	Counts retired Instructions that experienced STLB (2nd level TLB) true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_4	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 4 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_8	Counts retired instructions that are delivered to the back end after a front-end stall of at least 8 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_16	Counts retired instructions that are delivered to the back end after a front-end stall of at least 16 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_32	Counts retired instructions that are delivered to the back end after a front-end stall of at least 32 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_64	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 64 cycles which was not interrupted by a back-end stall.	Precise event capable.

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_128	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 128 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_256	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 256 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_512	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 512 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_1	Counts retired instructions that are delivered to the back end after the front end had at least 1 bubble-slot for a period of 2 cycles. A bubble-slot is an empty issue-pipeline slot while there was no RAT stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_2	Retired instructions that are fetched after an interval where the front end had at least 2 bubble-slots for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_3	Retired instructions that are fetched after an interval where the front end had at least 3 bubble-slots for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C7H	01H	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of SSE/AVX computational scalar double precision floating-point instructions retired. Each count represents 1 computation. Applies to SSE* and AVX* scalar double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT FM(N)ADD/SUB. FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as one DP FLOP.
C7H	02H	FP_ARITH_INST_RETIRED.SCALAR_SINGLE	Number of SSE/AVX computational scalar single precision floating-point instructions retired. Each count represents 1 computation. Applies to SSE* and AVX* scalar single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT FM(N)ADD/SUB. FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as one SP FLOP.
C7H	04H	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of SSE/AVX computational 128-bit packed double precision floating-point instructions retired. Each count represents 2 computations. Applies to SSE* and AVX* packed double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as two DP FLOPs.

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	08H	FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	Number of SSE/AVX computational 128-bit packed single precision floating-point instructions retired. Each count represents 4 computations. Applies to SSE* and AVX* packed single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as four SP FLOPs.
C7H	10H	FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of SSE/AVX computational 256-bit packed double precision floating-point instructions retired. Each count represents 4 computations. Applies to SSE* and AVX* packed double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as four DP FLOPs.
C7H	20H	FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	Number of SSE/AVX computational 256-bit packed single precision floating-point instructions retired. Each count represents 8 computations. Applies to SSE* and AVX* packed single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as eight SP FLOPs.
C7H	40H	FP_ARITH_INST_RETIRED.512B_PACKED_DOUBLE	Number of Packed Double-Precision FP arithmetic instructions (use operation multiplier of 8).	Only applicable when AVX-512 is enabled.
C7H	80H	FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE	Number of Packed Single-Precision FP arithmetic instructions (use operation multiplier of 16).	Only applicable when AVX-512 is enabled.
C8H	01H	HLE_RETIRED.START	Number of times we entered an HLE region. Does not count nested transactions.	
C8H	02H	HLE_RETIRED.COMMIT	Number of times HLE commit succeeded.	
C8H	04H	HLE_RETIRED.ABORTED	Number of times HLE abort was triggered.	Precise event capable.
C8H	08H	HLE_RETIRED.ABORTED_MEM	Number of times an HLE execution aborted due to various memory events (e.g., read/write capacity and conflicts).	
C8H	10H	HLE_RETIRED.ABORTED_TIMER	Number of times an HLE execution aborted due to hardware timer expiration.	
C8H	20H	HLE_RETIRED.ABORTED_UNFRIENDLY	Number of times an HLE execution aborted due to HLE-unfriendly instructions and certain unfriendly events (such as AD assists etc.).	
C8H	40H	HLE_RETIRED.ABORTED_MEMTYPE	Number of times an HLE execution aborted due to incompatible memory type.	
C8H	80H	HLE_RETIRED.ABORTED_EVENTS	Number of times an HLE execution aborted due to unfriendly events (such as interrupts).	
C9H	01H	RTM_RETIRED.START	Number of times we entered an RTM region. Does not count nested transactions.	
C9H	02H	RTM_RETIRED.COMMIT	Number of times RTM commit succeeded.	
C9H	04H	RTM_RETIRED.ABORTED	Number of times RTM abort was triggered.	Precise event capable.

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C9H	08H	RTM_RETIRED.ABORTED_MEM	Number of times an RTM execution aborted due to various memory events (e.g. read/write capacity and conflicts).	
C9H	10H	RTM_RETIRED.ABORTED_TIMER	Number of times an RTM execution aborted due to uncommon conditions.	
C9H	20H	RTM_RETIRED.ABORTED_UNFRIENDLY	Number of times an RTM execution aborted due to HLE-unfriendly instructions.	
C9H	40H	RTM_RETIRED.ABORTED_MEMTYPE	Number of times an RTM execution aborted due to incompatible memory type.	
C9H	80H	RTM_RETIRED.ABORTED_EVENTS	Number of times an RTM execution aborted due to none of the previous 4 categories (e.g. interrupt).	
CAH	1EH	FP_ASSIST.ANY	Counts cycles with any input and output SSE or x87 FP assist. If an input and output assist are detected on the same cycle the event increments by 1.	CounterMask=1 CMSK1
CBH	01H	HW_INTERRUPTS.RECEIVED	Counts the number of hardware interruptions received by the processor.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Increments when an entry is added to the Last Branch Record (LBR) array (or removed from the array in case of RETURNs in call stack mode). The event requires LBR enable via IA32_DEBUGCTL MSR and branch type selection via MSR_LBR_SELECT.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_4	Counts loads when the latency from first dispatch to completion is greater than 4 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_8	Counts loads when the latency from first dispatch to completion is greater than 8 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_16	Counts loads when the latency from first dispatch to completion is greater than 16 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_32	Counts loads when the latency from first dispatch to completion is greater than 32 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_64	Counts loads when the latency from first dispatch to completion is greater than 64 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_128	Counts loads when the latency from first dispatch to completion is greater than 128 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_256	Counts loads when the latency from first dispatch to completion is greater than 256 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_512	Counts loads when the latency from first dispatch to completion is greater than 512 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
D0H	11H	MEM_INST_RETIRED.STLB_MISS_LOADS	Retired load instructions that miss the STLB.	Precise event capable. PSDLA

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D0H	12H	MEM_INST_RETIRED.STLB_MISS_STORES	Retired store instructions that miss the STLB.	Precise event capable. PSDLA
D0H	21H	MEM_INST_RETIRED.LOCK_LOADS	Retired load instructions with locked access.	Precise event capable. PSDLA
D0H	41H	MEM_INST_RETIRED.SPLIT_LOADS	Counts retired load instructions that split across a cacheline boundary.	Precise event capable. PSDLA
D0H	42H	MEM_INST_RETIRED.SPLIT_STORES	Counts retired store instructions that split across a cacheline boundary.	Precise event capable. PSDLA
D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	Precise event capable. PSDLA
D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	Precise event capable. PSDLA
D1H	01H	MEM_LOAD_RETIRED.L1_HIT	Counts retired load instructions with at least one uop that hit in the L1 data cache. This event includes all SW prefetches and lock instructions regardless of the data source.	Precise event capable. PSDLA
D1H	02H	MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources.	Precise event capable. PSDLA
D1H	04H	MEM_LOAD_RETIRED.L3_HIT	Counts retired load instructions with at least one uop that hit in the L3 cache.	Precise event capable. PSDLA
D1H	08H	MEM_LOAD_RETIRED.L1_MISS	Counts retired load instructions with at least one uop that missed in the L1 cache.	Precise event capable. PSDLA
D1H	10H	MEM_LOAD_RETIRED.L2_MISS	Retired load instructions missed L2 cache as data sources.	Precise event capable. PSDLA
D1H	20H	MEM_LOAD_RETIRED.L3_MISS	Counts retired load instructions with at least one uop that missed in the L3 cache.	Precise event capable. PSDLA
D1H	40H	MEM_LOAD_RETIRED.FB_HIT	Counts retired load instructions with at least one uop was load missed in L1 but hit FB (Fill Buffers) due to preceding miss to the same cache line with data not ready.	Precise event capable. PSDLA
D2H	01H	MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS	Retired load instructions which data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Precise event capable. PSDLA
D2H	02H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT	Retired load instructions which data sources were L3 and cross-core snoop hits in on-pkg core cache.	Precise event capable. PSDLA
D2H	04H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM	Retired load instructions which data sources were HitM responses from shared L3.	Precise event capable. PSDLA
D2H	08H	MEM_LOAD_L3_HIT_RETIRED.XSNP_NONE	Retired load instructions which data sources were hits in L3 without snoops required.	Precise event capable. PSDLA
D3H	01H	MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM	Retired load instructions which data sources missed L3 but serviced from local DRAM.	Precise event capable.
D3H	02H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_DRAM	Retired load instructions which data sources missed L3 but serviced from remote dram.	Precise event capable.
D3H	04H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_HITM	Retired load instructions whose data sources was remote HITM.	Precise event capable.
D3H	08H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_FWD	Retired load instructions whose data sources was forwarded from a remote cache.	

Table 19-3. Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D4H	04H	MEM_LOAD_MISC_RETIRED.UC	Retired instructions with at least 1 uncacheable load or lock.	Precise event capable.
E6H	01H	BACLEARS.ANY	Counts the number of times the front-end is resteered when it finds a branch instruction in a fetch line. This occurs for the first time a branch instruction is fetched or when the branch is not tracked by the BPU (Branch Prediction Unit) anymore.	
F0H	40H	L2_TRANS.L2_WB	Counts L2 writebacks that access L2 cache.	
F1H	1FH	L2_LINES_IN.ALL	Counts the number of L2 cache lines filling the L2. Counting does not cover rejects.	
F2H	01H	L2_LINES_OUT.SILENT	Counts the number of lines that are silently dropped by L2 cache when triggered by an L2 cache fill. These lines are typically in Shared state. A non-threaded event.	
F2H	02H	L2_LINES_OUT.NON_SILENT	Counts the number of lines that are evicted by L2 cache when triggered by an L2 cache fill. Those lines can be either in modified state or clean state. Modified lines may either be written back to L3 or directly written to memory and not allocated in L3. Clean lines may either be allocated in L3 or dropped.	
F2H	04H	L2_LINES_OUT.USELESS_PREF	Counts the number of lines that have been hardware prefetched but not used and now evicted by L2 cache.	
F2H	04H	L2_LINES_OUT.USELESS_HWPF	Counts the number of lines that have been hardware prefetched but not used and now evicted by L2 cache.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of cache line split locks sent to the uncore.	
FEH	02H	IDI_MISC.WB_UPGRADE	Counts number of cache lines that are allocated and written back to L3 with the intention that they are more likely to be reused shortly.	
FEH	04H	IDI_MISC.WB_DOWNGRADE	Counts number of cache lines that are dropped and not written back to L3 as they are deemed to be less likely to be reused shortly.	

19.3 PERFORMANCE MONITORING EVENTS FOR 6TH GENERATION INTEL® CORE™ PROCESSOR AND 7TH GENERATION INTEL® CORE™ PROCESSOR

6th Generation Intel® Core™ processors are based on the Skylake microarchitecture. They support the architectural performance monitoring events listed in Table 19-1. Fixed counters in the core PMU support the architecture events defined in Table 19-2. Model-specific performance monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_4EH and 06_5EH. Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-5.

7th Generation Intel® Core™ processors are based on the Kaby Lake microarchitecture. Model-specific performance monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_8EH and 06_9EH.

The comment column in Table 19-4 uses abbreviated letters to indicate additional conditions applicable to the Event Mask Mnemonic. For event umasks listed in Table 19-4 that do not show "AnyT", users should refrain from programming "AnyThread =1" in IA32_PERF_EVTSELx.

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Load misses in all TLB levels causes a page walk that completes. (All page sizes.)	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a load.	
08H	10H	DTLB_LOAD_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a walk for a load.	CMSK1
08H	20H	DTLB_LOAD_MISSES.STLB_HIT	Loads that miss the DTLB but hit STLB.	
0DH	01H	INT_MISC.RECOVERY_CYCLES	Core cycles the allocator was stalled due to recovery from earlier machine clear event for this thread (for example, misprediction or memory order conflict).	
0DH	01H	INT_MISC.RECOVERY_CYCLES_ANY	Core cycles the allocator was stalled due to recovery from earlier machine clear event for any logical thread in this processor core.	AnyT
0DH	80H	INT_MISC.CLEAR_RESTEER_CYCLES	Cycles the issue-stage is waiting for front end to fetch from resteered path following branch misprediction or machine clear events.	
0EH	01H	UOPS_ISSUED.ANY	The number of uops issued by the RAT to RS.	
0EH	01H	UOPS_ISSUED.STALL_CYCLES	Cycles when the RAT does not issue uops to RS for the thread.	CMSK1, INV
0EH	02H	UOPS_ISSUED.VECTOR_WIDTH_MISMATCH	Uops inserted at issue-stage in order to preserve upper bits of vector registers.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
14H	01H	ARITH.FPU_DIVIDER_ACTIVE	Cycles when divider is busy executing divide or square root operations. Accounts for FP operations including integer divides.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand Data Read requests that missed L2, no rejects.	
24H	22H	L2_RQSTS.RFO_MISS	RFO requests that missed L2.	
24H	24H	L2_RQSTS.CODE_RD_MISS	L2 cache misses when fetching instructions.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that missed L2.	

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_L3_MISS_DEMAND_DATA_RD	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD_GE_6	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK6
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path.	
79H	04H	IDQ.MITE_CYCLES	Cycles when uops are being delivered to IDQ from MITE path.	CMSK1
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path.	
79H	08H	IDQ.DSB_CYCLES	Cycles when uops are being delivered to IDQ from DSB path.	CMSK1
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ by DSB when MS_busy.	
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Cycles DSB is delivered at least one uops.	CMSK1
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Cycles DSB is delivered four uops.	CMSK4
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ by MITE when MS_busy.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops.	CMSK1
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops.	CMSK4
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ while MS is busy.	
79H	30H	IDQ.MS_SWITCHES	Number of switches from DSB or MITE to MS.	EDG
79H	30H	IDQ.MS_CYCLES	Cycles MS is delivered at least one uops.	CMSK1
80H	04H	ICACHE_16B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache miss.	
80H	04H	ICACHE_64B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	
83H	01H	ICACHE_64B.IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	02H	ICACHE_64B.IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses at all ITLB levels that cause page walks.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Counts completed page walks in any TLB level due to code fetch misses (all page sizes).	
85H	10H	ITLB_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for an instruction fetch request.	

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	20H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOP_DELIV.CORE	Cycles which 4 issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	CMSK4
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_n_UOP_DELIV.CORE	Cycles which "4-n" issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	Set CMSK = 4-n; n = 1, 2, 3
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	Cycles which front end delivered 4 uops or the RAT was stalling FE.	CMSK, INV
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	
A2H	01H	RESOURCE_STALLS.ANY	Resource-related stall cycles.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_MISS	Cycles while L2 cache miss demand load is outstanding.	CMSK1
A3H	02H	CYCLE_ACTIVITY.CYCLES_L3_MISS	Cycles while L3 cache miss demand load is outstanding.	CMSK2
A3H	04H	CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls.	CMSK4
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_MISS	Execution stalls while L2 cache miss demand load is outstanding.	CMSK5
A3H	06H	CYCLE_ACTIVITY.STALLS_L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	CMSK6
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_MISS	Cycles while L1 data cache miss demand load is outstanding.	CMSK8
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_MISS	Execution stalls while L1 data cache miss demand load is outstanding.	CMSK12

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	10H	CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load.	CMSK16
A3H	14H	CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load.	CMSK20
A6H	01H	EXE_ACTIVITY.EXE_BOUND_0_PORTS	Cycles for which no uops began execution, the Reservation Station was not empty, the Store Buffer was full and there was no outstanding load.	
A6H	02H	EXE_ACTIVITY.1_PORTS_UTIL	Cycles for which one uop began execution on any port, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.2_PORTS_UTIL	Cycles for which two uops began execution, and the Reservation Station was not empty.	
A6H	08H	EXE_ACTIVITY.3_PORTS_UTIL	Cycles for which three uops began execution, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.4_PORTS_UTIL	Cycles for which four uops began execution, and the Reservation Station was not empty.	
A6H	40H	EXE_ACTIVITY.BOUND_ON_STORES	Cycles where the Store Buffer was full and no outstanding load.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
A8H	01H	LSD.CYCLES_ACTIVE	Cycles with at least one uop delivered by the LSD and none from the decoder.	CMSK1
A8H	01H	LSD.CYCLES_4_UOPS	Cycles with 4 uops delivered by the LSD and none from the decoder.	CMSK4
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	DSB-to-MITE switch true penalty cycles.	
AEH	01H	ITLB.ITLB_FLUSH	Flushing of the Instruction TLB (ITLB) pages, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B0H	10H	OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD	Demand data read requests that missed L3.	
B0H	80H	OFFCORE_REQUESTS.ALL_REQUESTS	Any memory transaction that reached the SQ.	
B1H	01H	UOPS_EXECUTED.THREAD	Counts the number of uops that begin execution across all ports.	
B1H	01H	UOPS_EXECUTED.STALL_CYCLES	Cycles where there were no uops that began execution.	CMSK, INV
B1H	01H	UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC	Cycles where there was at least one uop that began execution.	CMSK1
B1H	01H	UOPS_EXECUTED.CYCLES_GE_2_UOP_EXEC	Cycles where there were at least two uops that began execution.	CMSK2

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B1H	01H	UOPS_EXECUTED.CYCLES_GE_3_UOP_EXEC	Cycles where there were at least three uops that began execution.	CMSK3
B1H	01H	UOPS_EXECUTED.CYCLES_GE_4_UOP_EXEC	Cycles where there were at least four uops that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE	Counts the number of uops from any logical processor in this core that begin execution.	
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_1	Cycles where there was at least one uop, from any logical processor in this core, that began execution.	CMSK1
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_2	Cycles where there were at least two uops, from any logical processor in this core, that began execution.	CMSK2
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_3	Cycles where there were at least three uops, from any logical processor in this core, that began execution.	CMSK3
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_4	Cycles where there were at least four uops, from any logical processor in this core, that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_NONE	Cycles where there were no uops from any logical processor in this core that began execution.	CMSK1, INV
B1H	10H	UOPS_EXECUTED.X87	Counts the number of X87 uops that begin execution.	
B2H	01H	OFF_CORE_REQUEST_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this core.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	01H	TLB_FLUSH.STLB_ANY	STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only;
C0H	01H	INST_RETIRED.TOTAL_CYCLES	Number of cycles using always true condition applied to PEBS instructions retired event.	CMSK10, PS
C1H	3FH	OTHER_ASSISTS.ANY	Number of times a microcode assist is invoked by HW other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists.	
C2H	01H	UOPS_RETIRED.STALL_CYCLES	Cycles without actually retired uops.	CMSK1, INV
C2H	01H	UOPS_RETIRED.TOTAL_CYCLES	Cycles with less than 10 actually retired uops.	CMSK10, INV
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Retirement slots used.	
C3H	01H	MACHINE_CLEARS.COUNT	Number of machine clears of any type.	CMSK1, EDG
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions that retired.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	PS

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	PS
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	PS
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	PS
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	PS
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	PS
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were mispredicted and taken.	PS
C6H	01H	FRONTEND_RETIRED.DSB_MISS	Retired instructions which experienced DSB miss. Specify MSR_PEBS_FRONTEND.EVTSEL=11H.	PS
C6H	01H	FRONTEND_RETIRED.L1I_MISS	Retired instructions which experienced instruction L1 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=12H.	PS
C6H	01H	FRONTEND_RETIRED.L2_MISS	Retired instructions which experienced L2 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=13H.	PS
C6H	01H	FRONTEND_RETIRED.ITLB_MISS	Retired instructions which experienced ITLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=14H.	PS
C6H	01H	FRONTEND_RETIRED.STLB_MISS	Retired instructions which experienced STLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=15H.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_16	Retired instructions that are fetched after an interval where the front end delivered no uops for at least 16 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =16, IDQ_Bubble_Width = 4.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_m	Retired instructions that are fetched after an interval where the front end had 'm' IDQ slots delivered, no uops for at least 2 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =2, IDQ_Bubble_Width = m.	PS, m = 1, 2, 3
C7H	01H	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of double-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one DP FLOP.
C7H	02H	FP_ARITH_INST_RETIRED.SCALAR_SINGLE	Number of single-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one SP FLOP.

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	04H	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of double-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPD instruction counts as 2.	Software may treat each count as two DP FLOPs.
C7H	08H	FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	Number of single-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPS instruction counts as 2.	Software may treat each count as four SP FLOPs.
C7H	10H	FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of double-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA instruction counts as 2.	Software may treat each count as four DP FLOPs.
C7H	20H	FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	Number of single-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA or VDPPS instruction counts as 2.	Software may treat each count as eight SP FLOPs.
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	CMSK1
CBH	01H	HW_INTERRUPTS.RECEIVED	Number of hardware interrupts received by the processor.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Increments when an entry is added to the Last Branch Record (LBR) array (or removed from the array in case of RETURNs in call stack mode). The event requires LBR enable via IA32_DEBUGCTL MSR and branch type selection via MSR_LBR_SELECT.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PSDLA
D0H	11H	MEM_INST_RETIRED.STLB_MISS_LOADS	Retired load instructions that miss the STLB.	PSDLA
D0H	12H	MEM_INST_RETIRED.STLB_MISS_STORES	Retired store instructions that miss the STLB.	PSDLA
D0H	21H	MEM_INST_RETIRED.LOCK_LOADS	Retired load instructions with locked access.	PSDLA
D0H	41H	MEM_INST_RETIRED.SPLIT_LOADS	Number of load instructions retired with cache-line splits that may impact performance.	PSDLA
D0H	42H	MEM_INST_RETIRED.SPLIT_STORES	Number of store instructions retired with line-split.	PSDLA
D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	PSDLA
D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	PSDLA
D1H	01H	MEM_LOAD_RETIRED.L1_HIT	Retired load instructions with L1 cache hits as data sources.	PSDLA
D1H	02H	MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources.	PSDLA
D1H	04H	MEM_LOAD_RETIRED.L3_HIT	Retired load instructions with L3 cache hits as data sources.	PSDLA
D1H	08H	MEM_LOAD_RETIRED.L1_MISS	Retired load instructions missed L1 cache as data sources.	PSDLA
D1H	10H	MEM_LOAD_RETIRED.L2_MISS	Retired load instructions missed L2. Unknown data source excluded.	PSDLA

Table 19-4. Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	20H	MEM_LOAD_RETIRED.L3_MISS	Retired load instructions missed L3. Excludes unknown data source.	PSDLA
D1H	40H	MEM_LOAD_RETIRED.FB_HIT	Retired load instructions where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	PSDLA
D2H	01H	MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS	Retired load instructions where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	PSDLA
D2H	02H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT	Retired load Instructions where data sources were L3 and cross-core snoop hits in on-pkg core cache.	PSDLA
D2H	04H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM	Retired load instructions where data sources were HitM responses from shared L3.	PSDLA
D2H	08H	MEM_LOAD_L3_HIT_RETIRED.XSNP_NONE	Retired load instructions where data sources were hits in L3 without snoops required.	PSDLA
E6H	01H	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	

CMSK1: Counter Mask = 1 required; CMSK4: CounterMask = 4 required; CMSK6: CounterMask = 6 required; CMSK8: CounterMask = 8 required; CMSK10: CounterMask = 10 required; CMSK12: CounterMask = 12 required; CMSK16: CounterMask = 16 required; CMSK20: CounterMask = 20 required.

AnyT: AnyThread = 1 required.

INV: Invert = 1 required.

EDG: EDGE = 1 required.

PSDLA: Also supports PEBS and DataLA.

PS: Also supports PEBS.

Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-5.

Table 19-5. Intel® TSX Performance Event Addendum in Processors based on Skylake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.4 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON PHI™ PROCESSOR 3200, 5200, 7200 SERIES

Intel® Xeon Phi™ processors 3200/5200/7200 series are based on the Knights Landing microarchitecture. Model-specific performance monitoring events in the processor core are listed in Table 19-6. The events in Table 19-6 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following value 06_57H.

Table 19-6. Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	RECYCLEQ.LD_BLOCK_ST_FORWARD	Counts the number of occurrences a retired load gets blocked because its address partially overlaps with a store.	PSDLA
03H	02H	RECYCLEQ.LD_BLOCK_STD_NOT READY	Counts the number of occurrences a retired load gets blocked because its address overlaps with a store whose data is not ready.	
03H	04H	RECYCLEQ.ST_SPLITS	Counts the number of occurrences a retired store that is a cache line split. Each split should be counted only once.	
03H	08H	RECYCLEQ.LD_SPLITS	Counts the number of occurrences a retired load that is a cache line split. Each split should be counted only once.	PSDLA
03H	10H	RECYCLEQ.LOCK	Counts all the retired locked loads. It does not include stores because we would double count if we count stores.	
03H	20H	RECYCLEQ.STA_FULL	Counts the store micro-ops retired that were pushed in the recycle queue because the store address buffer is full.	
03H	40H	RECYCLEQ.ANY_LD	Counts any retired load that was pushed into the recycle queue for any reason.	
03H	80H	RECYCLEQ.ANY_ST	Counts any retired store that was pushed into the recycle queue for any reason.	
04H	01H	MEM_UOPS_RETIRED.L1_MISS_ LOADS	Counts the number of load micro-ops retired that miss in L1 D cache.	
04H	02H	MEM_UOPS_RETIRED.L2_HIT_L OADS	Counts the number of load micro-ops retired that hit in the L2.	PSDLA
04H	04H	MEM_UOPS_RETIRED.L2_MISS_ LOADS	Counts the number of load micro-ops retired that miss in the L2.	PSDLA
04H	08H	MEM_UOPS_RETIRED.DTLB_MIS S_LOADS	Counts the number of load micro-ops retired that cause a DTLB miss.	PSDLA
04H	10H	MEM_UOPS_RETIRED.UTLB_MIS S_LOADS	Counts the number of load micro-ops retired that caused micro TLB miss.	
04H	20H	MEM_UOPS_RETIRED.HITM	Counts the loads retired that get the data from the other core in the same tile in M state.	
04H	40H	MEM_UOPS_RETIRED.ALL_LOAD S	Counts all the load micro-ops retired.	
04H	80H	MEM_UOPS_RETIRED.ALL_STOR ES	Counts all the store micro-ops retired.	
05H	01H	PAGE_WALKS.D_SIDE_WALKS	Counts the total D-side page walks that are completed or started. The page walks started in the speculative path will also be counted.	EdgeDetect=1
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts the total number of core cycles for all the D-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	02H	PAGE_WALKS.I_SIDE_WALKS	Counts the total I-side page walks that are completed.	EdgeDetect=1

Table 19-6. Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts the total number of core cycles for all the I-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	03H	PAGE_WALKS.WALKS	Counts the total page walks that are completed (I-side and D-side).	EdgeDetect=1
05H	03H	PAGE_WALKS.CYCLES	Counts the total number of core cycles for all the page walks. The cycles for page walks started in speculative path will also be included.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts the number of L2 cache misses. Also called L2_REQUESTS_MISS.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts the total number of L2 cache references. Also called L2_REQUESTS_REFERENCE.	
30H	00H	L2_REQUESTS_REJECT.ALL	Counts the number of MEC requests from the L2Q that reference a cache line (cacheable requests) excluding SW prefetches filling only to L2 cache and L1 evictions (automatically excludes L2HWP, UC, WC) that were rejected - Multiple repeated rejects should be counted multiple times.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of MEC requests that were not accepted into the L2Q because of any L2 queue reject condition. There is no concept of at-ret here. It might include requests due to instructions in the speculative path.	
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of unhalted core clock cycles.	
3CH	01H	CPU_CLK_UNHALTED.REF	Counts the number of unhalted reference clock cycles.	
3EH	04H	L2_PREFETCHER.ALLOC_XQ	Counts the number of L2HWP allocated into XQ GP.	
80H	01H	ICACHE.HIT	Counts all instruction fetches that hit the instruction cache.	
80H	02H	ICACHE.MISSES	Counts all instruction fetches that miss the instruction cache or produce memory requests. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	ICACHE.ACCESSES	Counts all instruction fetches, including uncacheable fetches.	
86H	04H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts the number of core cycles the fetch stalls because of an icache miss. This is a cumulative count of core cycles the fetch stalled for all icache misses.	
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.4.1.1.2.	Requires MSR_OFFCORE_RESP 0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.4.1.1.2.	Requires MSR_OFFCORE_RESP 1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Counts the total number of instructions retired.	PS

Table 19-6. Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C2H	01H	UOPS_RETIRED.MS	Counts the number of micro-ops retired that are from the complex flows issued by the micro-sequencer (MS).	
C2H	10H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired.	
C2H	20H	UOPS_RETIRED.SCALAR_SIMD	Counts the number of scalar SSE, AVX, AVX2, and AVX-512 micro-ops except for loads (memory-to-register mov-type micro ops), division and sqrt.	
C2H	40H	UOPS_RETIRED.PACKED_SIMD	Counts the number of packed SSE, AVX, AVX2, and AVX-512 micro-ops (both floating point and integer) except for loads (memory-to-register mov-type micro-ops), packed byte and word multiplies.	
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the machine clears due to program modifying data within 1K of a recently fetched code page.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of times the machine clears due to memory ordering hazards.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts the number of floating operations retired that required microcode assists.	
C3H	08H	MACHINE_CLEARS.ALL	Counts all machine clears.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	7EH	BR_INST_RETIRED.JCC	Counts the number of JCC branch instructions retired.	PS
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts the number of far branch instructions retired.	PS
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts the number of branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C4H	F7H	BR_INST_RETIRED.RETURN	Counts the number of near RET branch instructions retired.	PS
C4H	F9H	BR_INST_RETIRED.CALL	Counts the number of near CALL branch instructions retired.	PS
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts the number of near indirect CALL branch instructions retired.	PS
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts the number of near relative CALL branch instructions retired.	PS
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts the number of branch instructions retired that were taken conditional jumps.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts the number of mispredicted branch instructions retired.	PS
C5H	7EH	BR_MISP_RETIRED.JCC	Counts the number of mispredicted JCC branch instructions retired.	PS
C5H	BFH	BR_MISP_RETIRED.FAR_BRANCH	Counts the number of mispredicted far branch instructions retired.	PS
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts the number of mispredicted branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts the number of mispredicted near RET branch instructions retired.	PS

Table 19-6. Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	F9H	BR_MISP_RETIRED.CALL	Counts the number of mispredicted near CALL branch instructions retired.	PS
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts the number of mispredicted near indirect CALL branch instructions retired.	PS
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Counts the number of mispredicted near relative CALL branch instructions retired.	PS
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts the number of mispredicted branch instructions retired that were taken conditional jumps.	PS
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of core cycles when no micro-ops are allocated and the ROB is full.	
CAH	04H	NO_ALLOC_CYCLES.MISPREDICTS	Counts the number of core cycles when no micro-ops are allocated and the alloc pipe is stalled waiting for a mispredicted branch to retire.	
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of core cycles when no micro-ops are allocated and a RATstall (caused by reservation station full) is asserted.	
CAH	90H	NO_ALLOC_CYCLES.NOT_DELIVERED	Counts the number of core cycles when no micro-ops are allocated, the IQ is empty, and no other condition is blocking allocation.	
CAH	7FH	NO_ALLOC_CYCLES.ALL	Counts the total number of core cycles when no micro-ops are allocated for any reason.	
CBH	01H	RS_FULL_STALL.MEC	Counts the number of core cycles when allocation pipeline is stalled and is waiting for a free MEC reservation station entry.	
CBH	1FH	RS_FULL_STALL.ALL	Counts the total number of core cycles the allocation pipeline is stalled when any one of the reservation stations is full.	
CDH	01H	CYCLES_DIV_BUSY.ALL	Cycles the number of core cycles when divider is busy. Does not imply a stall waiting for the divider.	
E6H	01H	BACLEARS.ALL	Counts the number of times the front end resteers for any branch as a result of another branch handling mechanism in the front end.	
E6H	08H	BACLEARS.RETURN	Counts the number of times the front end resteers for RET branches as a result of another branch handling mechanism in the front end.	
E6H	10H	BACLEARS.COND	Counts the number of times the front end resteers for conditional branches as a result of another branch handling mechanism in the front end.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the MSROM starts a flow of uops.	

PS: Also supports PEBS.

PSDLA: Also supports PEBS and DataLA.

19.5 PERFORMANCE MONITORING EVENTS FOR THE INTEL® CORE™ M AND 5TH GENERATION INTEL® CORE™ PROCESSORS

The Intel® Core™ M processors, the 5th generation Intel® Core™ processors and the Intel Xeon processor E3 1200 v4 product family are based on the Broadwell microarchitecture. They support the architectural performance monitoring events listed in Table 19-1. Model-specific performance monitoring events in the processor core are listed in Table 19-7. The events in Table 19-7 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3DH and 06_47H. Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are available on processors based on Broadwell microarchitecture. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Model-specific performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Broadwell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3DH and 06_47H support uncore performance events listed in Table 19-11.

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles when divider is busy executing divide operations.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4FH	10H	EPT.WALK_CYCLES	Cycles of Extended Page Table walks.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding demand code read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	Number of uops delivered to IDQ from any path.	

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that cause a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	Set AnyThread to count per core.
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	Set AnyThread to count per core.
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	Set AnyThread to count per core.
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	Set AnyThread to count per core.
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	Set AnyThread to count per core.
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	Set AnyThread to count per core.
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	Set AnyThread to count per core.
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles of delay due to Decode Stream Buffer to MITE switches.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-logical-processor each cycle.	Use Cmask to count stall cycles.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C0H	02H	INST_RETIRED.X87	FP operations retired. X87 FP operations that have no exceptions.	
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	01H	MACHINE_CLEARS.CYCLES	Counts cycles while a machine clears stalled forward progress of a logical processor or a processor core.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.

Table 19-7. Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source.	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops where data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops where data sources were HitM responses from shared L3.	Supports PEBS and DataLA.
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops where data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops where data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	

Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Broadwell microarchitecture. Where Broadwell microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-8.

Table 19-8. Intel® TSX Performance Event Addendum in Processors Based on Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.6 PERFORMANCE MONITORING EVENTS FOR THE 4TH GENERATION INTEL® CORE™ PROCESSORS

4th generation Intel® Core™ processors and Intel Xeon processor E3-1200 v3 product family are based on the Haswell microarchitecture. They support the architectural performance monitoring events listed in Table 19-1. Model-specific performance monitoring events in the processor core are listed in Table 19-9. The events in Table 19-9 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3CH, 06_45H and 06_46H. Table 19-10 lists performance events focused on supporting Intel TSX (see Section 18.3.6.5). Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g., derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to demand load misses that caused 2M/4M page walks in any TLB levels.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Completed page walks in any TLB of any page size due to demand load misses.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
08H	40H	DTLB_LOAD_MISSES.STLB_HIT_2M	Load misses that missed DTLB but hit STLB (2M).	
08H	60H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
08H	80H	DTLB_LOAD_MISSES.PDE_CACHE_MISS	DTLB demand load misses with low part of linear-to-physical address translation missed.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1 to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	42H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache.	
24H	22H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	44H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	24H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that miss L2 cache.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	Demand requests to L2 cache.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
24H	3FH	L2_RQSTS.MISS	All requests that missed L2.	
24H	FFH	L2_RQSTS.REFERENCES	All requests to L2 cache.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to store misses in one or more TLB levels of 2M/4M page structure.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Completed page walks due to store miss in any TLB levels of any page size (4K/2M/4M/1G).	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
49H	40H	DTLB_STORE_MISSES.STLB_HIT_2M	Store misses that missed DTLB but hit STLB (2M).	
49H	60H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
49H	80H	DTLB_STORE_MISSES.PDE_CACHE_MISS	DTLB store misses with low part of linear-to-physical address translation missed.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that causes a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to misses in ITLB 2M/4M page entries.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Completed page walks in ITLB of any page size.	
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
85H	40H	ITLB_MISSES.STLB_HIT_2M	ITLB misses that hit STLB (2M).	
85H	60H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_EXECUTED_PORT.PORT_0	Cycles which a uop is dispatched on port 0 in this thread.	Set AnyThread to count per core.
A1H	02H	UOPS_EXECUTED_PORT.PORT_1	Cycles which a uop is dispatched on port 1 in this thread.	Set AnyThread to count per core.
A1H	04H	UOPS_EXECUTED_PORT.PORT_2	Cycles which a uop is dispatched on port 2 in this thread.	Set AnyThread to count per core.
A1H	08H	UOPS_EXECUTED_PORT.PORT_3	Cycles which a uop is dispatched on port 3 in this thread.	Set AnyThread to count per core.
A1H	10H	UOPS_EXECUTED_PORT.PORT_4	Cycles which a uop is dispatched on port 4 in this thread.	Set AnyThread to count per core.
A1H	20H	UOPS_EXECUTED_PORT.PORT_5	Cycles which a uop is dispatched on port 5 in this thread.	Set AnyThread to count per core.
A1H	40H	UOPS_EXECUTED_PORT.PORT_6	Cycles which a uop is dispatched on port 6 in this thread.	Set AnyThread to count per core.

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	80H	UOPS_EXECUTED_PORT.PORT_7	Cycles which a uop is dispatched on port 7 in this thread	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set Cmask=2 to count cycle.	Use only when HTT is off.
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set Cmask=2 to count cycle.	
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Use only when HTT is off.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 data cache miss loads. Set Cmask=8 to count cycle.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFF_CORE_RESPONSE_0	See Table 18-28 or Table 18-29.	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Table 18-28 or Table 18-29.	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
BCH	28H	PAGE_WALKER_LOADS.ITLB_MEMORY	Number of ITLB page walker loads from memory.	

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use Cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA; use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were taken but mispredicted.	

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source .	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared L3.	Supports PEBS and DataLA.

Table 19-9. Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	06H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	

Table 19-10. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	01H	TX_MEM.ABORT_CONFLICT	Number of times a transactional abort was signaled due to a data conflict on a transactionally accessed address.	
54H	02H	TX_MEM.ABORT_CAPACITY_WRITE	Number of times a transactional abort was signaled due to a data capacity limitation for transactional writes.	
54H	04H	TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a HLE transactional region aborted due to a non XRELEASE prefixed instruction writing to an elided lock in the elision buffer.	
54H	08H	TX_MEM.ABORT_HLE_ELISION_BUFFER_NOT_EMPTY	Number of times an HLE transactional execution aborted due to NoAllocatedElisionBuffer being non-zero.	
54H	10H	TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH	Number of times an HLE transactional execution aborted due to XRELEASE lock not satisfying the address and value requirements in the elision buffer.	
54H	20H	TX_MEM.ABORT_HLE_ELISION_BUFFER_UNSUPPORTED_ALIGNMENT	Number of times an HLE transactional execution aborted due to an unsupported read alignment from the elision buffer.	

Table 19-10. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	40H	TX_MEM.HLE_ELISION_BUFFER_FULL	Number of times HLE lock could not be elided due to ElisionBufferAvailable being zero.	
5DH	01H	TX_EXEC.MISC1	Counts the number of times a class of instructions that may cause a transactional abort was executed. Since this is the count of execution, it may not always cause a transactional abort.	
5DH	02H	TX_EXEC.MISC2	Counts the number of times a class of instructions (for example, vzeroupper) that may cause a transactional abort was executed inside a transactional region.	
5DH	04H	TX_EXEC.MISC3	Counts the number of times an instruction execution caused the transactional nest count supported to be exceeded.	
5DH	08H	TX_EXEC.MISC4	Counts the number of times an XBEGIN instruction was executed inside an HLE transactional region.	
5DH	10H	TX_EXEC.MISC5	Counts the number of times an instruction with HLE-XACQUIRE semantic was executed inside an RTM transactional region.	
C8H	01H	HLE_RETIRED.START	Number of times an HLE execution started.	IF HLE is supported.
C8H	02H	HLE_RETIRED.COMMIT	Number of times an HLE execution successfully committed.	
C8H	04H	HLE_RETIRED.ABORTED	Number of times an HLE execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	
C8H	08H	HLE_RETIRED.ABORTED_MEM	Number of times an HLE execution aborted due to various memory events (for example, read/write capacity and conflicts).	
C8H	10H	HLE_RETIRED.ABORTED_TIMER	Number of times an HLE execution aborted due to uncommon conditions.	
C8H	20H	HLE_RETIRED.ABORTED_UNFRIENDLY	Number of times an HLE execution aborted due to HLE-unfriendly instructions.	
C8H	40H	HLE_RETIRED.ABORTED_MEMTYPE	Number of times an HLE execution aborted due to incompatible memory type.	
C8H	80H	HLE_RETIRED.ABORTED_EVENTS	Number of times an HLE execution aborted due to none of the previous 4 categories (for example, interrupts).	
C9H	01H	RTM_RETIRED.START	Number of times an RTM execution started.	IF RTM is supported.
C9H	02H	RTM_RETIRED.COMMIT	Number of times an RTM execution successfully committed.	
C9H	04H	RTM_RETIRED.ABORTED	Number of times an RTM execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	

Table 19-10. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C9H	08H	RTM_RETIRED.ABORTED_MEM	Number of times an RTM execution aborted due to various memory events (for example, read/write capacity and conflicts).	IF RTM is supported.
C9H	10H	RTM_RETIRED.ABORTED_TIMER	Number of times an RTM execution aborted due to uncommon conditions.	
C9H	20H	RTM_RETIRED.ABORTED_UNFRIENDLY	Number of times an RTM execution aborted due to HLE-unfriendly instructions.	
C9H	40H	RTM_RETIRED.ABORTED_MEMTYPE	Number of times an RTM execution aborted due to incompatible memory type.	
C9H	80H	RTM_RETIRED.ABORTED_EVENTS	Number of times an RTM execution aborted due to none of the previous 4 categories (for example, interrupt).	

Model-specific performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Haswell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3CH and 06_45H support performance events listed in Table 19-11.

Table 19-11. Uncore Performance Events in the 4th Generation Intel® Core™ Processors

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to L3 eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	L3 lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	06H	UNC_CBO_CACHE_LOOKUP.ES	L3 lookup request that access cache and found line in E or S state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	L3 lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	

Table 19-11. Uncore Performance Events in the 4th Generation Intel® Core™ Processors (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or L3.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or L3.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and L3 evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of L3 evictions allocated.	
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.6.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v3 Family

Model-specific performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v3 family based on the Haswell-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3FH, are listed in Table 19-12. The performance events listed in Table 19-9 and Table 19-10 also apply Intel Xeon processor E5 v3 family, except that the OFF_CORE_RESPONSE_x event listed in Table 19-9 should reference Table 18-30.

Uncore performance monitoring events for Intel Xeon Processor E5 v3 families are described in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual".

Table 19-12. Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 v3 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	04H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data sources were remote DRAM (snoop not needed, Snoop Miss).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote cache HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.7 PERFORMANCE MONITORING EVENTS FOR 3RD GENERATION INTEL® CORE™ PROCESSORS

3rd generation Intel® Core™ processors and Intel Xeon processor E3-1200 v2 product family are based on Intel microarchitecture code name Ivy Bridge. They support architectural performance monitoring events listed in Table 19-1. Model-specific performance monitoring events in the processor core are listed in Table 19-13. The events in Table 19-13 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3AH. Fixed counters in the core PMU support the architecture events defined in Table 19-24.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at found at https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	81H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size from demand loads.	
08H	82H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size by demand loads.	
08H	84H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk due to demand loads.	
08H	88H	DTLB_LOAD_MISSES.LARGE_PAGE_WALK_DURATION	Page walk for a large page completed for Demand load.	
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops adds delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* or AVX-128 double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* or AVX-128 single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* or AVX-128 single precision FP packed uops executed.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* or AVX-128 double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks that missed LLC.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache lines in any state.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer Move Elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD Move Elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer Move Elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD Move Elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
5FH	04H	DTLB_LOAD_MISSES.STLB_HIT	Counts load operations that missed 1st level DTLB but hit the 2nd level.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand Code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	04H	ICACHE.IFETCH_STALL	Cycles where a code-fetch stalled due to L1 instruction-cache miss or an iTLB miss.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_EXECUTE	Cycles of dispatch stalls. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Restricted to counters 0-3 when HTT is disabled.
A3H	06H	CYCLE_ACTIVITY.STALLS_LDM_PENDING		Restricted to counters 0-3 when HTT is disabled.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFFCORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	80H	OTHER_ASSISTS.WB	Number of times microcode assist is invoked by hardware upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	Supports PEBS.
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	Supports PEBS.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	Supports PEBS.
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	Supports PEBS.
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	Supports PEBS.
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PMC 3 only.
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.3.4.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops whose data source followed an L1 miss.	Supports PEBS.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops that missed L2, excluding unknown sources.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops whose data source is LLC miss.	Supports PEBS. Restricted to counters 0-3 when HTT is disabled.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data source was local memory (cross-socket snoop not needed or missed).	Supports PEBS.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	

Table 19-13. Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or LLC HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by the MLC prefetcher.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by the MLC prefetcher.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.

19.7.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v2 Family and Intel Xeon Processor E7 v2 Family

Model-specific performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v2 family and Intel Xeon processor E7 v2 family based on the Ivy Bridge-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3EH, are listed in Table 19-14.

Table 19-14. Performance Events Applicable Only to the Processor Core of Intel® Xeon® Processor E5 v2 Family and Intel® Xeon® Processor E7 v2 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	03H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data sources were local DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	0CH	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data source was remote DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.8 PERFORMANCE MONITORING EVENTS FOR 2ND GENERATION INTEL® CORE™ I7-2XXX, INTEL® CORE™ I5-2XXX, INTEL® CORE™ I3-2XXX PROCESSOR SERIES

2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel Xeon processor E3-1200 product family are based on the Intel microarchitecture code name Sandy Bridge. They support architectural performance monitoring events listed in Table 19-1. Model-specific performance monitoring events in the processor core are listed in Table 19-15, Table 19-16, and Table 19-17. The events in Table 19-15 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_2AH and 06_2DH. The events in Table 19-16 apply to processors with CPUID signature 06_2AH. The events in Table 19-17 apply to processors with CPUID signature 06_2DH. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at found at https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Blocked loads due to store buffer blocks with unknown data.	
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	# of Split loads blocked due to resource not available.	
03H	10H	LD_BLOCKS.ALL_BLOCK	Number of cases where any load is blocked but has no DCU miss.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
07H	08H	LD_BLOCKS_PARTIAL.ALL_STA_BLOCK	The number of times that load operations are temporarily blocked because of older stores, with addresses that are not yet known. A load operation may incur more than one block of this type.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size.	
08H	04H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears or JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0DH	40H	INT_MISC.RAT_STALL_CYCLES	Cycles RAT external stall is sent to IDQ for this thread.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
17H	01H	INSTS_WRITTEN_TO_IQ.INSTS	Counts the number of instructions written into the IQ every cycle.	
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Requests from L2 Hardware prefetcher that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Requests from L2 Hardware prefetcher that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Any requests from L2 Hardware prefetchers.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	04H	L2_STORE_LOCK_RQSTS.HIT_E	RFOs that hit cache lines in E state.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks from L1D to L2 cache lines that missed L2.	
28H	02H	L2_L1D_WB_RQSTS.HIT_S	Not rejected writebacks from L1D to L2 cache lines in S state.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4EH	02H	HW_PRE_REQ.DL1_MISS	Hardware Prefetch requests that miss the L1D cache. A request is being counted each time it access the cache & miss it, including if a block is applicable or if hit the Fill Buffer for example.	This accounts for both L1 streamer and IP-based (IPP) HW prefetchers.
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
51H	02H	L1D.ALLOCATED_IN_M	Counts the number of allocations of modified L1D cache lines.	
51H	04H	L1D.EVICTION	Counts the number of modified lines evicted from the L1 data cache due to replacement.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
51H	08H	L1D.ALL_M_REPLACEMENT	Cache lines in M state evicted out of L1D due to Snoop HitM or dirty line replacement.	
59H	20H	PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP	Increments the number of flags-merge uops in flight each cycle. Set Cmask = 1 to count cycles.	
59H	40H	PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW	Cycles with at least one slow LEA uop allocated.	
59H	80H	PARTIAL_RAT_STALLS.MUL_SINGLE_UOP	Number of Multiply packed/scalar single precision uops allocated.	
5BH	0CH	RESOURCE_STALLS2.ALL_FL_EMPTY	Cycles stalled due to free list empty.	PMC0-3 only regardless HTT.
5BH	0FH	RESOURCE_STALLS2.ALL_PRF_CONTROL	Cycles stalled due to control structures full for physical registers.	
5BH	40H	RESOURCE_STALLS2.BOB_FULL	Cycles Allocator is stalled due Branch Order Buffer.	
5BH	4FH	RESOURCE_STALLS2.OOO_RSRC	Cycles stalled due to out of order resources full.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS busy by DSB. Set Cmask = 1 to count cycles MS is busy. Set Cmask=1 and Edge =1 to count MS activations.	Can combine Umask 08H and 10H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS is busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H and 30H.
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	41H	BR_INST_EXEC.NONTAKEN_CONDITIONAL	Not-taken macro conditional branches.	
88H	81H	BR_INST_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired conditional branches.	
88H	82H	BR_INST_EXEC.TAKEN_DIRECT_JUMP	Taken speculative and retired conditional branches excluding calls and indirects.	
88H	84H	BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired indirect branches excluding calls and returns.	
88H	88H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN	Taken speculative and retired indirect branches that are returns.	
88H	90H	BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired direct near calls.	
88H	A0H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired indirect near calls.	
88H	C1H	BR_INST_EXEC.ALL_CONDITIONAL	Speculative and retired conditional branches.	
88H	C2H	BR_INST_EXEC.ALL_DIRECT_JUMP	Speculative and retired conditional branches excluding calls and indirects.	
88H	C4H	BR_INST_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired indirect branches excluding calls and returns.	
88H	C8H	BR_INST_EXEC.ALL_INDIRECT_NEAR_RETURN	Speculative and retired indirect branches that are returns.	
88H	D0H	BR_INST_EXEC.ALL_NEAR_CALL	Speculative and retired direct near calls.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Speculative and retired branches.	
89H	41H	BR_MISP_EXEC.NONTAKEN_CONDITIONAL	Not-taken mispredicted macro conditional branches.	
89H	81H	BR_MISP_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired mispredicted conditional branches.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
89H	84H	BR_MISP_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	88H	BR_MISP_EXEC.TAKEN_RETURN_NEAR	Taken speculative and retired mispredicted indirect branches that are returns.	
89H	90H	BR_MISP_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired mispredicted direct near calls.	
89H	A0H	BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired mispredicted indirect near calls.	
89H	C1H	BR_MISP_EXEC.ALL_CONDITIONAL	Speculative and retired mispredicted conditional branches.	
89H	C4H	BR_MISP_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	D0H	BR_MISP_EXEC.ALL_NEAR_CALL	Speculative and retired mispredicted direct near calls.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Speculative and retired mispredicted branches.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	02H	RESOURCE_STALLS.LB	Counts the cycles of stall due to lack of load buffers.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FCSW	Cycles stalled due to writing the FPU control word.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	
A3H	02H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_DISPATCH	Cycles of dispatch stalls. Set AnyThread to count per core.	PMC0-3 only.

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	05H	CYCLE_ACTIVITY.STALL_CYCLES_L2_PENDING		PMC0-3 only.
A3H	06H	CYCLE_ACTIVITY.STALL_CYCLES_L1D_PENDING		PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	02H	DSB_FILL.OTHER_CANCEL	Cases of cancelling valid DSB fill not because of exceeding way limit.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_DISPATCHED.THREAD	Counts total number of uops to be dispatched per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	PMC0-3 only regardless HTT.
B1H	02H	UOPS_DISPATCHED.CORE	Counts total number of uops to be dispatched per-core each cycle.	Do not need to set ANY.
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this thread core.	
B6H	01H	AGU_BYPASS_CANCEL.COUNT	Counts executed load operations with all the following traits: 1. Addressing of the format [base + offset], 2. The offset is between 1 and 2047, 3. The address specified in the base register is in one page and the address [base+offset] is in another page.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
BFH	05H	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports.	Cmask=1.
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only; must quiesce other PMCs.
C1H	02H	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	10H	BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 assists due to output value.	

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 assists due to input value.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization. PMC3 only.	Specify threshold in MSR 3F6H.
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.3.4.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops which data sources were data hits in LLC without snoops required.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops which data sources were data missed LLC (excluding unknown data source).	Supports PEBS.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.

Table 19-15. Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
E6H	01H	BACLEARS.ANY	Counts the number of times the front end is re-steered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front end.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	L2 or LLC HW prefetches that access L2 cache.	Including rejects.
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by L2 prefetch.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by L2 prefetch.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.
F4H	10H	SQ_MISC.SPLIT_LOCK	Split locks in SQ.	

Non-architecture performance monitoring events in the processor core that are applicable only to Intel processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH are listed in Table 19-16.

Table 19-16. Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were LLC hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS. PMC0-3 only regardless HTT.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were LLC and cross-core snoop hits in on-pkg core cache.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared LLC.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in LLC without snoops required.	Supports PEBS.

Table 19-16. Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D4H	02H	MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS	Retired load uops with unknown information as data source in cache serviced the load.	Supports PEBS. PMC0-3 only regardless HTT.
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT_N		10003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.MISS_DRAM_N		300400244H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0091H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_MISS.DRAM_N		300400091H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_MISS.DRAM_N		300400240H
		OFFCORE_RESPONSE.ALL_PF_DATA_RD.LLC_MISS.DRAM_N		300400090H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0120H
		OFFCORE_RESPONSE.ALL_PF_RfO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.SNOOP_MISS_N		2003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_MISS.DRAM_N		300400120H
		OFFCORE_RESPONSE.ALL_READS.LLC_MISS.DRAM_N		3004003F7H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.SNOOP_MISS_N		2003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_MISS.DRAM_N		300400122H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.DRAM_N		300400004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.DRAM_N		300400001H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0002H

Table 19-16. Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.SNOOP_MISS_N		2003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_MISS.DRAM_N		300400002H
		OFFCORE_RESPONSE.OTHER.ANY_RESPONSE_N		18000H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.DRAM_N		300400040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.DRAM_N		300400010H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.SNOOP_MISS_N		2003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_MISS.DRAM_N		300400020H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.DRAM_N		300400200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.DRAM_N		300400080H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.SNOOP_MISS_N		2003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_MISS.DRAM_N		300400100H

Non-architecture performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 family (and Intel Core i7-3930 processor) based on Intel microarchitecture code name Sandy Bridge, with CPUID signature of DisplayFamily_DisplayModel 06_2DH, are listed in Table 19-17.

Table 19-17. Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Additional Configuration: Disable BL bypass and direct2core, and if the memory is remotely homed. The count is not reliable If the memory is locally homed.	
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Additional Configuration: Disable BL bypass. Supports PEBS.	

Table 19-17. Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Additional Configuration: Disable BL bypass and direct2core. Supports PEBS.	
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Additional Configuration: Disable bypass. Supports PEBS.	
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources were data missed LLC but serviced by local DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
D3H	04H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops which data sources were data missed LLC but serviced by remote DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.LOCAL_DRAM_N		600400004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_DRAM_N		67F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HITM_N		107FC00004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00001H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00010H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00080H

Model-specific performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Sandy Bridge. Processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH support performance events listed in Table 19-18.

Table 19-18. Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to LLC eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	LLC lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	02H	UNC_CBO_CACHE_LOOKUP.E	LLC lookup request that access cache and found line in E-state.	
34H	04H	UNC_CBO_CACHE_LOOKUP.S	LLC lookup request that access cache and found line in S-state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	LLC lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or LLC.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or LLC.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and LLC evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of LLC evictions allocated.	

Table 19-18. Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.9 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ I7 PROCESSOR FAMILY AND INTEL® XEON® PROCESSOR FAMILY

Processors based on the Intel microarchitecture code name Nehalem support the architectural and model-specific performance monitoring events listed in Table 19-1 and Table 19-19. The events in Table 19-19 generally applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_1AH, 06_1EH, 06_1FH, and 06_2EH. However, Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a small number of events that are not supported in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. These events are noted in the comment column.

In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, 06_1FH) also support the following model-specific, product-specific uncore performance monitoring events listed in Table 19-20.

Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	07H	SB_DRAIN.ANY	Counts the number of store buffer drains.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
08H	80H	DTLB_LOAD_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to load miss in the STLB.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set "invert=1, cmask = 1".
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	
0FH	01H	MEM_UNCORE_RETIRED.L3_DATA_MISS_UNKNOWN	Counts number of memory load instructions retired where the memory reference missed L3 and data source is unknown.	Available only for CPUID signature 06_2EH.
0FH	02H	MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM	Counts number of memory load instructions retired where the memory reference hit modified data in a sibling core residing on the same socket.	
0FH	08H	MEM_UNCORE_RETIRED.REMOTE_CACHE_LOCAL_HOME_HIT	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and HIT in a remote socket's cache. Only counts locally homed lines.	
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and was remotely homed. This includes both DRAM access and HITM in a remote socket's cache for remotely homed lines.	
0FH	20H	MEM_UNCORE_RETIRED.LOCAL_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and required a local socket memory reference. This includes locally homed cachelines that were in a modified state in another socket.	
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and to perform I/O.	Available only for CPUID signature 06_2EH.

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Counts number of loops that can't stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, for example, a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	
2EH	4FH	L3_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache. The event count includes speculative traffic but excludes cache line fills due to a L2 hardware-prefetch. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	41H	L3_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache. The event count may include speculative traffic but excludes cache line fills due to L2 hardware-prefetches. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
40H	01H	L1D_CACHE_LD.I_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the I (invalid) state, i.e. the read request missed the cache.	Counter 0, 1 only.
40H	02H	L1D_CACHE_LD.S_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.
40H	04H	L1D_CACHE_LD.E_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
40H	08H	L1D_CACHE_LD.M_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
40H	0FH	L1D_CACHE_LD.MESI	Counts L1 data cache read requests.	Counter 0, 1 only.
41H	02H	L1D_CACHE_ST.S_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.
41H	04H	L1D_CACHE_ST.E_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
41H	08H	L1D_CACHE_ST.M_STATE	Counts L1 data cache store RFO requests where cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
42H	01H	L1D_CACHE_LOCK.HIT	Counts retired load locks that hit in the L1 data cache or hit in an already allocated fill buffer. The lock portion of the load lock transaction must hit in the L1D.	The initial load will pull the lock into the L1 data cache. Counter 0, 1 only.
42H	02H	L1D_CACHE_LOCK.S_STATE	Counts L1 data cache retired load locks that hit the target cache line in the shared state.	Counter 0, 1 only.
42H	04H	L1D_CACHE_LOCK.E_STATE	Counts L1 data cache retired load locks that hit the target cache line in the exclusive state.	Counter 0, 1 only.

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
42H	08H	L1D_CACHE_LOCK.M_STATE	Counts L1 data cache retired load locks that hit the target cache line in the modified state.	Counter 0, 1 only.
43H	01H	L1D_ALL_REF.ANY	Counts all references (uncached, speculated and retired) to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once.	The event does not include non-memory accesses, such as I/O accesses. Counter 0, 1 only.
43H	02H	L1D_ALL_REF.CACHEABLE	Counts all data reads and writes (speculated and retired) from cacheable memory, including locked operations.	Counter 0, 1 only.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk for large pages.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
53H	01H	L1D_CACHE_LOCK_FB_HIT	Counts the number of cacheable load lock speculated or retired instructions accepted into the fill buffer.	
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts cycles when the uops executed were issued from any ports except port 5. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts cycles when the uops are executing. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)".	Requires programming MSR 01A7H.

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of MMX instructions retired.	
C0H	04H	INST_RETIRED.MMX	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating point Uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating point Uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating point Uops retired.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating point Uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer Uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_PORT	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe. Cycles when partial register stalls occurred. Cycles when flag stalls occurred. Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating-point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	

Table 19-19. Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions (denormal input when the DAZ flag is off or underflow result when the FTZ flag is off); x87 instructions (NaN or denormal are loaded to a register or used as input from memory, division by 0 or underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Model-specific performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Nehalem. Processors with CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, and 06_1FH support performance events listed in Table 19-20.

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
20H	01H	UNC_QHL_REQUESTS.IOH_READS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WRITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE_READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE_WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
27H	01H	UNC_QMC_NORMAL_FULL.READ.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with read requests.	
27H	02H	UNC_QMC_NORMAL_FULL.READ.CH1	Uncore cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with read requests.	
27H	04H	UNC_QMC_NORMAL_FULL.READ.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with read requests.	
27H	08H	UNC_QMC_NORMAL_FULL.WRITE.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with write requests.	
27H	10H	UNC_QMC_NORMAL_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with write requests.	
27H	20H	UNC_QMC_NORMAL_FULL.WRITE.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with write requests.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTO PRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTO PRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	
63H	20H	UNC_DRAM_READ_CAS.AUTO PRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	
64H	02H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	

Table 19-20. Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
64H	20H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	

Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a distinct uncore sub-system that is significantly different from the uncore found in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. Model-specific performance monitoring events for its uncore will be available in future documentation.

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

Intel 64 processors based on Intel® microarchitecture code name Westmere support the architectural and model-specific performance monitoring events listed in Table 19-1 and Table 19-21. Table 19-21 applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_25H, 06_2CH. In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH) also support the following model-specific, product-specific uncore performance monitoring events listed in Table 19-22. Fixed counters support the architecture events defined in Table 19-2.

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LOAD_BLOCK.OVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAIN.ANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORY.STORE	All store referenced with misaligned address.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	04H	DTLB_LOAD_MISSES.WALK_CYCLES	Cycles PMH is busy with a page walk due to a load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set “invert=1, cmask = 1“.
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0FH	01H	MEM_UNCORE_RETIRED.UNKNOWN_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets.
0FH	02H	MEM_UNCORE_RETIRED.OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets.
0FH	04H	MEM_UNCORE_RETIRED.REMOTE_HITM	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only.
0FH	08H	MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets.
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM (Precise Event).	Applicable to two sockets only.
0FH	20H	MEM_UNCORE_RETIRED.OTHER_LLC_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only.
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets.
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Number of loops that cannot stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	
2EH	41H	L3_LAT_CACHE.MISS	Counts uncore Last Level Cache misses. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	4FH	L3_LAT_CACHE.REFERENCE	Counts uncore Last Level Cache references. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	04H	DTLB_MISSES.WALK_CYCLES	Counts cycles of page walk due to misses in the STLB.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	Counter 0, 1 only.

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	Counter 0, 1 only.
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	Counter 0, 1 only.
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	Counter 0, 1 only.
4FH	10H	EPT.WALK_CYCLES	Counts Extended Page walk cycles.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_DATA	Counts weighted cycles of offcore demand data read requests. Does not include L2 prefetch requests.	Counter 0.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_CODE	Counts weighted cycles of offcore demand code read requests. Does not include L2 prefetch requests.	Counter 0.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.RFO	Counts weighted cycles of offcore demand RFO requests. Does not include L2 prefetch requests.	Counter 0.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ANY.READ	Counts weighted cycles of offcore read requests of any kind. Include L2 prefetch requests.	Counter 0.
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table. This event does not cause locks, it merely detects them.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
85H	04H	ITLB_MISSES.WALK_CYCLES	Counts ITLB miss page walk cycles.	
85H	10H	ITLB_MISSES.STLB_HIT	Counts number of ITLB first level miss but second level hits.	
85H	80H	ITLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	
B0H	01H	OFFCORE_REQUESTS.DEMAND.READ_DATA	Counts number of offcore demand data read requests. Does not count L2 prefetch requests.	
B0H	02H	OFFCORE_REQUESTS.DEMAND.READ_CODE	Counts number of offcore demand code read requests. Does not count L2 prefetch requests.	
B0H	04H	OFFCORE_REQUESTS.DEMAND.RFO	Counts number of offcore demand RFO requests. Does not count L2 prefetch requests.	
B0H	08H	OFFCORE_REQUESTS.ANY.READ	Counts number of offcore read requests. Includes L2 prefetch requests.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	10H	OFFCORE_REQUESTS.ANY.RFO	Counts number of offcore RFO requests. Includes L2 prefetch requests.	
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B0H	80H	OFFCORE_REQUESTS.ANY	Counts all offcore requests.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts number of cycles there are one or more uops being executed and were issued on ports 0-4. This is a core count only and cannot be collected per thread.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts number of cycles there are one or more uops being executed on any ports. This is a core count only and cannot be collected per thread.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B3H	01H	SNOOPQ_REQUESTS_OUTSTANDING.DATA	Counts weighted cycles of snoopq requests for data. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	02H	SNOOPQ_REQUESTS_OUTSTANDING.INVALIDATE	Counts weighted cycles of snoopq invalidate requests. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	04H	SNOOPQ_REQUESTS_OUTSTANDING.CODE	Counts weighted cycles of snoopq requests for code. Counter 0 only.	Use cmask=1 to count cycles not empty.
B4H	01H	SNOOPQ_REQUESTS.CODE	Counts the number of snoop code requests.	
B4H	02H	SNOOPQ_REQUESTS.DATA	Counts the number of snoop data requests.	
B4H	04H	SNOOPQ_REQUESTS.INVALIDATE	Counts the number of snoop invalidate requests.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Use MSR 01A7H.
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C0H	04H	INST_RETIRED.MMX	Counts the number of retired: MMX instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Counts mispredicted conditional retired calls.	
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Counts all mispredicted retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating-point uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating-point uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating-point uops retired.	
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating-point uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	
D1H	01H	UOPS_DECODED.STALL_CYCLES	Counts the cycles of decoder stalls. INV=1, Cmask=1.	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_PORT	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe, Cycles when partial register stalls occurred, Cycles when flag stalls occurred, Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
ECH	01H	THREAD_ACTIVE	Counts cycles threads are active.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	04H	SQ_MISC.LRU_HINTS	Counts number of Super Queue LRU hints sent to L3.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions, (Denormal input when the DAZ flag is off or Underflow result when the FTZ flag is off): x87 instructions, (NaN or denormal are loaded to a register or used as input from memory, Division by 0 or Underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	

Table 19-21. Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Model-specific performance monitoring events of the uncore sub-system for processors with CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH, and 06_1FH support performance events listed in Table 19-22.

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
02H	01H	UNC_GQ_OCCUPANCY.READ_TRACKER	Increments the number of queue entries (code read, data read, and RFOs) in the tread tracker. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
0CH	01H	UNC_GQ_SNOOP.GOTO_S	Counts the number of remote snoops that have requested a cache line be set to the S state.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0CH	02H	UNC_GQ_SNOOP.GOTO_I	Counts the number of remote snoops that have requested a cache line be set to the I state.	
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the S state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the S state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the S state from M state.	Requires writing MSR 301H with mask = 1H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the S state from S state.	Requires writing MSR 301H with mask = 4H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the I state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the I state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the I state from M state.	Requires writing MSR 301H with mask = 1H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the I state from S state.	Requires writing MSR 301H with mask = 4H.
20H	01H	UNC_QHL_REQUESTS.IOH_READS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WRITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE_READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE_WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2AH	07H	UNC_QMC_OCCUPANCY.ANY	Normal read request occupancy for any channel.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
32H	01H	UNC_IMC_RETRY.CH0	Counts number of IMC DRAM channel 0 retries. DRAM retry only occurs when configured in RAS mode.	
32H	02H	UNC_IMC_RETRY.CH1	Counts number of IMC DRAM channel 1 retries. DRAM retry only occurs when configured in RAS mode.	
32H	04H	UNC_IMC_RETRY.CH2	Counts number of IMC DRAM channel 2 retries. DRAM retry only occurs when configured in RAS mode.	
32H	07H	UNC_IMC_RETRY.ANY	Counts number of IMC DRAM retries from any channel. DRAM retry only occurs when configured in RAS mode.	
33H	01H	UNC_QHL_FRC_ACK_CNFLTS.IOH	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the IOH.	
33H	02H	UNC_QHL_FRC_ACK_CNFLTS.REMOTE	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the remote home.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	
33H	07H	UNC_QHL_FRC_ACK_CNFLTS.ANY	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic.	
34H	01H	UNC_QHL_SLEEPS.IOH_ORDER	Counts number of occurrences a request was put to sleep due to IOH ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	02H	UNC_QHL_SLEEPS.REMOTE_ORDER	Counts number of occurrences a request was put to sleep due to remote socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	04H	UNC_QHL_SLEEPS.LOCAL_ORDER	Counts number of occurrences a request was put to sleep due to local socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
34H	08H	UNC_QHL_SLEEPS.IOH_CONFLICT	Counts number of occurrences a request was put to sleep due to IOH address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	10H	UNC_QHL_SLEEPS.REMOTE_CONFLICT	Counts number of occurrences a request was put to sleep due to remote socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	20H	UNC_QHL_SLEEPS.LOCAL_CONFLICT	Counts number of occurrences a request was put to sleep due to local socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
35H	01H	UNC_ADDR_OPCODE_MATCH.IOH	Counts number of requests from the IOH, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	02H	UNC_ADDR_OPCODE_MATCH.REMOTE	Counts number of requests from the remote socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	04H	UNC_ADDR_OPCODE_MATCH.LOCAL	Counts number of requests from the local socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	01H	UNC_QPI_TX_HEADER.FULL.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is full.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	04H	UNC_QPI_TX_HEADER.FULL.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is full.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTOPRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTOPRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	20H	UNC_DRAM_READ_CAS.AUTO PRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	
64H	02H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	
64H	20H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
67H	01H	UNC_DRAM_THERMAL_THROT TLED	Uncore cycles DRAM was throttled due to its temperature being above the thermal throttling threshold.	
80H	01H	UNC_THERMAL_THROTTLING_ TEMP.CORE_0	Cycles that the PCU records that core 0 is above the thermal throttling threshold temperature.	

Table 19-22. Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	UNC_THERMAL_THROTTLING_TEMP.CORE_1	Cycles that the PCU records that core 1 is above the thermal throttling threshold temperature.	
80H	04H	UNC_THERMAL_THROTTLING_TEMP.CORE_2	Cycles that the PCU records that core 2 is above the thermal throttling threshold temperature.	
80H	08H	UNC_THERMAL_THROTTLING_TEMP.CORE_3	Cycles that the PCU records that core 3 is above the thermal throttling threshold temperature.	
81H	01H	UNC_THERMAL_THROTTLED_TEMP.CORE_0	Cycles that the PCU records that core 0 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	02H	UNC_THERMAL_THROTTLED_TEMP.CORE_1	Cycles that the PCU records that core 1 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	04H	UNC_THERMAL_THROTTLED_TEMP.CORE_2	Cycles that the PCU records that core 2 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	08H	UNC_THERMAL_THROTTLED_TEMP.CORE_3	Cycles that the PCU records that core 3 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
82H	01H	UNC_PROCHOT_ASSERTION	Number of system assertions of PROCHOT indicating the entire processor has exceeded the thermal limit.	
83H	01H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_0	Cycles that the PCU records that core 0 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	02H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_1	Cycles that the PCU records that core 1 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	04H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_2	Cycles that the PCU records that core 2 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	08H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_3	Cycles that the PCU records that core 3 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
84H	01H	UNC_TURBO_MODE.CORE_0	Uncore cycles that core 0 is operating in turbo mode.	
84H	02H	UNC_TURBO_MODE.CORE_1	Uncore cycles that core 1 is operating in turbo mode.	
84H	04H	UNC_TURBO_MODE.CORE_2	Uncore cycles that core 2 is operating in turbo mode.	
84H	08H	UNC_TURBO_MODE.CORE_3	Uncore cycles that core 3 is operating in turbo mode.	
85H	02H	UNC_CYCLES_UNHALTED_L3_FLL_ENABLE	Uncore cycles that at least one core is unhalted and all L3 ways are enabled.	
86H	01H	UNC_CYCLES_UNHALTED_L3_FLL_DISABLE	Uncore cycles that at least one core is unhalted and all L3 ways are disabled.	

19.11 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 5200, 5400 SERIES AND INTEL® CORE™2 EXTREME PROCESSORS QX 9000 SERIES

Processors based on the Enhanced Intel Core microarchitecture support the architectural and model-specific performance monitoring events listed in Table 19-1 and Table 19-25. In addition, they also support the following model-specific performance monitoring events listed in Table 19-23. Fixed counters support the architecture events defined in Table 19-24.

Table 19-23. Performance Events for Processors Based on Enhanced Intel Core Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	08H	INST_RETIRED.VM_HOST	Instruction retired while in VMX root operations.	
D2H	10H	RAT_STAALS.OTHER_SERIALIZATION_STALLS	This event counts the number of stalls due to other RAT resource serialization not counted by Umask value 0FH.	

19.12 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 3000, 3200, 5100, 5300 SERIES AND INTEL® CORE™2 DUO PROCESSORS

Processors based on the Intel® Core™ microarchitecture support architectural and model-specific performance monitoring events.

Fixed-function performance counters are introduced first on processors based on Intel Core microarchitecture. Table 19-24 lists pre-defined performance events that can be counted using fixed-function performance counters.

Table 19-24. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
MSR_PERF_FIXED_CTR1/IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.CORE	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time. When the core frequency is constant, this event can approximate elapsed time while the core was not in halt state.
MSR_PERF_FIXED_CTR2/IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF	This event counts the number of reference cycles when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction.

Table 19-24. Fixed-Function Performance Counter and Pre-defined Performance Events (Contd.)

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
			This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in halt state and not in a TM stop-clock state. This event has a constant ratio with the CPU_CLK_UNHALTED.BUS event.

Table 19-25 lists general-purpose model-specific performance monitoring events supported in processors based on Intel® Core™ microarchitecture. For convenience, Table 19-25 also includes architectural events and describes minor model-specific behavior where applicable. Software must use a general-purpose performance counter to count events listed in Table 19-25.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture

Event Num	Umask Value	Event Name	Definition	Description and Comment
03H	02H	LOAD_BLOCK.STA	Loads blocked by a preceding store with unknown address.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to an address that is not yet calculated. The number of events is greater or equal to the number of load operations that were blocked. If the load and the store are always to different addresses, check why the memory disambiguation mechanism is not working. To avoid such blocks, increase the distance between the store and the following load so that the store address is known at the time the load is dispatched.
03H	04H	LOAD_BLOCK.STD	Loads blocked by a preceding store with unknown data.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to the same address and the stored data value is not yet known. The number of events is greater or equal to the number of load operations that were blocked. To avoid such blocks, increase the distance between the store and the dependent load, so that the store data is known at the time the load is dispatched.
03H	08H	LOAD_BLOCK. OVERLAP_STORE	Loads that partially overlap an earlier store, or 4-Kbyte aliased with a previous store.	This event indicates that loads are blocked due to a variety of reasons. Some of the triggers for this event are when a load is blocked by a preceding store, in one of the following: ▪ Some of the loaded byte locations are written by the preceding store and some are not. ▪ The load is from bytes written by the preceding store, the store is aligned to its size and either: ▪ The load's data size is one or two bytes and it is not aligned to the store. ▪ The load's data size is of four or eight bytes and the load is misaligned.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				▪ The load is from bytes written by the preceding store, the store is misaligned and the load is not aligned on the beginning of the store. ▪ The load is split over an eight byte boundary (excluding 16-byte loads). ▪ The load and store have the same offset relative to the beginning of different 4-KByte pages. This case is also called 4-KByte aliasing. ▪ In all these cases the load is blocked until after the blocking store retires and the stored data is committed to the cache hierarchy.
03H	10H	LOAD_BLOCK. UNTIL_RETIRE	Loads blocked until retirement.	This event indicates that load operations were blocked until retirement. The number of events is greater or equal to the number of load operations that were blocked. This includes mainly uncacheable loads and split loads (loads that cross the cache line boundary) but may include other cases where loads are blocked until retirement.
03H	20H	LOAD_BLOCK.L1D	Loads blocked by the L1 data cache.	This event indicates that loads are blocked due to one or more reasons. Some triggers for this event are: ▪ The number of L1 data cache misses exceeds the maximum number of outstanding misses supported by the processor. This includes misses generated as result of demand fetches, software prefetches or hardware prefetches. ▪ Cache line split loads. ▪ Partial reads, such as reads to un-cacheable memory, I/O instructions and more. ▪ A locked load operation is in progress. The number of events is greater or equal to the number of load operations that were blocked.
04H	01H	SB_DRAIN_ CYCLES	Cycles while stores are blocked due to store buffer drain.	This event counts every cycle during which the store buffer is draining. This includes: ▪ Serializing operations such as CPUID ▪ Synchronizing operations such as XCHG ▪ Interrupt acknowledgment ▪ Other conditions, such as cache flushing
04H	02H	STORE_BLOCK. ORDER	Cycles while store is waiting for a preceding store to be globally observed.	This event counts the total duration, in number of cycles, which stores are waiting for a preceding stored cache line to be observed by other cores. This situation happens as a result of the strong store ordering behavior, as defined in "Memory Ordering," Chapter 8, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. The stall may occur and be noticeable if there are many cases when a store either misses the L1 data cache or hits a cache line in the Shared state. If the store requires a bus transaction to read the cache line then the stall ends when snoop response for the bus transaction arrives.
04H	08H	STORE_BLOCK. SNOOP	A store is blocked due to a conflict with an external or internal snoop.	This event counts the number of cycles the store port was used for snooping the L1 data cache and a store was stalled by the snoop. The store is typically resubmitted one cycle later.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
06H	00H	SEGMENT_REG_ LOADS	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.
07H	00H	SSE_PRE_EXEC. NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
07H	01H	SSE_PRE_EXEC.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	02H	SSE_PRE_EXEC.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.
07H	03H	SSE_PRE_ EXEC.STORES	Streaming SIMD Extensions (SSE) Weakly-ordered store instructions executed.	This event counts the number of times SSE non-temporal store instructions are executed.
08H	01H	DTLB_MISSES. ANY	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	02H	DTLB_MISSES .MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	04H	DTLB_MISSES.L0_MISS_LD	L0 DTLB misses due to load operations.	This event counts the number of level 0 Data Table Lookaside Buffer (DTLB0) misses due to load operations. This count includes misses detected as a result of speculative accesses. Loads that miss that DTLB0 and hit the DTLB1 can incur two-cycle penalty.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
08H	08H	DTLB_MISSES. MISS_ST	TLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses. Address translation for store operations is performed in the DTLB1.
09H	01H	MEMORY_ DISAMBIGUATION.RESET	Memory disambiguation reset cycles.	This event counts the number of cycles during which memory disambiguation misprediction occurs. As a result the execution pipeline is cleaned and execution of the mispredicted load instruction and all succeeding instructions restarts. This event occurs when the data address accessed by a load instruction, collides infrequently with preceding stores, but usually there is no collision. It happens rarely, and may have a penalty of about 20 cycles.
09H	02H	MEMORY_DISAMBIGUATIO N.SUCCESS	Number of loads successfully disambiguated.	This event counts the number of load operations that were successfully disambiguated. Loads are preceded by a store with an unknown address, but they are not blocked.
0CH	01H	PAGE_WALKS .COUNT	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. The average can hint whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
0CH	02H	PAGE_WALKS. CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. The average can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
10H	00H	FP_COMP_OPS _EXE	Floating point computational micro-ops executed.	This event counts the number of floating point computational micro-ops executed. Use IA32_PMC0 only.
11H	00H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: ▪ Streaming SIMD Extensions (SSE) instructions: ▪ Denormal input when the DAZ (Denormals Are Zeros) flag is off ▪ Underflow result when the FTZ (Flush To Zero) flag is off ▪ X87 instructions: ▪ NaN or denormal are loaded to a register or used as input from memory ▪ Division by 0 ▪ Underflow output Use IA32_PMC1 only.
12H	00H	MUL	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations. Use IA32_PMC1 only.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
13H	00H	DIV	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed. Use IA32_PMC1 only.
14H	00H	CYCLES_DIV _BUSY	Cycles the divider busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Use IA32_PMC0 only.
18H	00H	IDLE_DURING _DIV	Cycles the divider is busy and all other execution units are idle.	This event counts the number of cycles the divider is busy (with a divide or a square root operation) and no other execution unit or load operation is in progress. Load operations are assumed to hit the L1 data cache. This event considers only micro-ops dispatched after the divider started operating. Use IA32_PMC0 only.
19H	00H	DELAYED_ BYPASS.FP	Delayed bypass to FP operation.	This event counts the number of times floating point operations use data immediately after the data was generated by a non-floating point execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	01H	DELAYED_ BYPASS.SIMD	Delayed bypass to SIMD operation.	This event counts the number of times SIMD operations use data immediately after the data was generated by a non-SIMD execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	02H	DELAYED_ BYPASS.LOAD	Delayed bypass to load operation.	This event counts the number of delayed bypass penalty cycles that a load operation incurred. When load operations use data immediately after the data was generated by an integer execution unit, they may (pending on certain dynamic internal conditions) incur one penalty cycle due to delayed data bypass between the units. Use IA32_PMC1 only.
21H	See Table 18-61	L2_ADS.(Core)	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. It can count occurrences for this core or both cores.
23H	See Table 18-61	L2_DBUS_BUSY _RD.(Core)	Cycles the L2 transfers data to the core.	This event counts the number of cycles during which the L2 data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. This event can count occurrences for this core or both cores.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
24H	Combined mask from Table 18-61 and Table 18-63	L2_LINES_IN. (Core, Prefetch)	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. It can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-61	L2_M_LINES_IN. (Core)	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.
26H	See Table 18-61 and Table 18-63	L2_LINES_OUT. (Core, Prefetch)	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-61 and Table 18-63	L2_M_LINES_OUT.(Core, Prefetch)	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a modified-state in one of the L1 data caches. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
28H	Combined mask from Table 18-61 and Table 18-64	L2_IFETCH.(Core, Cache Line State)	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the IFU. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
29H	Combined mask from Table 18-61, Table 18-63, and Table 18-64	L2_LD.(Core, Prefetch, Cache Line State)	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. The event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together or separately. ▪ Of accesses to cache lines at different MESI states.
2AH	See Table 18-61 and Table 18-64	L2_ST.(Core, Cache Line State)	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
2BH	See Table 18-61 and Table 18-64	L2_LOCK.(Core, Cache Line State)	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
2EH	See Table 18-61, Table 18-63, and Table 18-64	L2_RQSTS.(Core, Prefetch, Cache Line State)	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
2EH	41H	L2_RQSTS.SELF. DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF. DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
30H	See Table 18-61, Table 18-63, and Table 18-64	L2_REJECT_BUSQ.(Core, Prefetch, Cache Line State)	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: ▪ The bus queue is full. ▪ The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
32H	See Table 18-61	L2_NO_REQ.(Core)	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests were pending from a core. When using the BOTH_CORE modifier, the event counts only if none of the cores have a pending request. The event counts also when one core is halted and the other is not halted. The event can count occurrences for this core or both cores.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep Technology (EIST) transitions.	This event counts the number of transitions that include a frequency change, either with or without voltage change. This includes Enhanced Intel SpeedStep Technology (EIST) and TM2 transitions. The event is incremented only while the counting core is in C0 state. Since transitions to higher-numbered CxE states and TM2 transitions include a frequency change or voltage transition, the event is incremented accordingly.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond and returns to normal when the temperature falls below the thermal trip threshold temperature.
3CH	00H	CPU_CLK_ UNHALTED. CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason, this event may have a changing ratio in regard to time. When the core frequency is constant, this event can give approximate elapsed time while the core not in halt state. This is an architectural performance event.
3CH	01H	CPU_CLK_ UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give a measurement of the elapsed time while the core was not in the halt state. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_ UNHALTED.NO _OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	See Table 18-64	L1D_CACHE_LD. (Cache Line State)	L1 cacheable data reads.	This event counts the number of data reads from cacheable memory. Locked reads are not counted.
41H	See Table 18-64	L1D_CACHE_ST. (Cache Line State)	L1 cacheable data writes.	This event counts the number of data writes to cacheable memory. Locked writes are not counted.
42H	See Table 18-64	L1D_CACHE_ LOCK.(Cache Line State)	L1 data cacheable locked reads.	This event counts the number of locked data reads from cacheable memory.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
42H	10H	L1D_CACHE_ LOCK_DURATION	Duration of L1 data cacheable locked operation.	This event counts the number of cycles during which any cache line is locked by any locking instruction. Locking happens at retirement and therefore the event does not occur for instructions that are speculatively executed. Locking duration is shorter than locked instruction execution duration.
43H	01H	L1D_ALL_REF	All references to the L1 data cache.	This event counts all references to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once. The event includes non-cacheable accesses, such as I/O accesses.
43H	02H	L1D_ALL_ CACHE_REF	L1 Data cacheable reads and writes.	This event counts the number of data reads and writes from cacheable memory, including locked operations. This event is a sum of: ▪ L1D_CACHE_LD.MESI ▪ L1D_CACHE_ST.MESI ▪ L1D_CACHE_LOCK.MESI
45H	0FH	L1D_REPL	Cache lines allocated in the L1 data cache.	This event counts the number of lines brought into the L1 data cache.
46H	00H	L1D_M_REPL	Modified cache lines allocated in the L1 data cache.	This event counts the number of modified lines brought into the L1 data cache.
47H	00H	L1D_M_EVICT	Modified cache lines evicted from the L1 data cache.	This event counts the number of modified lines evicted from the L1 data cache, whether due to replacement or by snoop HITM intervention.
48H	00H	L1D_PEND_ MISS	Total number of outstanding L1 data cache misses at any cycle.	This event counts the number of outstanding L1 data cache misses at any cycle. An L1 data cache miss is outstanding from the cycle on which the miss is determined until the first chunk of data is available. This event counts: ▪ All cacheable demand requests. ▪ L1 data cache hardware prefetch requests. ▪ Requests to write through memory. ▪ Requests to write combine memory. Uncacheable requests are not counted. The count of this event divided by the number of L1 data cache misses, L1D_REPL, is the average duration in core cycles of an L1 data cache miss.
49H	01H	L1D_SPLIT.LOADS	Cache line split loads from the L1 data cache.	This event counts the number of load operations that span two cache lines. Such load operations are also called split loads. Split load operations are executed at retirement.
49H	02H	L1D_SPLIT. STORES	Cache line split stores to the L1 data cache.	This event counts the number of store operations that span two cache lines.
4BH	00H	SSE_PRE_ MISS.NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchNTA were executed and missed all cache levels. Due to speculation an executed instruction might not retire. This instruction prefetches the data to the L1 data cache.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
4BH	01H	SSE_PRE_ MISS.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT0 were executed and missed all cache levels. Due to speculation executed instruction might not retire. The prefetchT0 instruction prefetches data to the L2 cache and L1 data cache.
4BH	02H	SSE_PRE_ MISS.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 were executed and missed all cache levels. Due to speculation, an executed instruction might not retire. The prefetchT1 and PrefetchNT2 instructions prefetch data to the L2 cache.
4CH	00H	LOAD_HIT_PRE	Load operations conflicting with a software prefetch to the same address.	This event counts load operations sent to the L1 data cache while a previous Streaming SIMD Extensions (SSE) prefetch instruction to the same cache line has started prefetching but has not yet finished.
4EH	10H	L1D_PREFETCH. REQUESTS	L1 data cache prefetch requests.	This event counts the number of times the L1 data cache requested to prefetch a data cache line. Requests can be rejected when the L2 cache is busy and resubmitted later or lost. All requests are counted, including those that are rejected.
60H	See Table 18-61 and Table 18-62.	BUS_REQUEST_ OUTSTANDING. (Core and Bus Agents)	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. The event counts only full-line cacheable read requests from either the L1 data cache or the L2 prefetchers. It does not count Read for Ownership transactions, instruction byte fetch transactions, or any other bus transaction.
61H	See Table 18-62.	BUS_BNR_DRV. (Bus Agents)	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. To obtain the number of bus cycles during which the BNR signal is asserted, multiply the event count by two. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance.
62H	See Table 18-62.	BUS_DRDY_ CLOCKS.(Bus Agents)	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. With the 'THIS_AGENT' mask this event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. With the 'ALL_AGENTS' mask, this event counts the number of bus cycles during which any bus agent sends data on the bus. This includes all data reads and writes on the bus.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
63H	See Table 18-61 and Table 18-62.	BUS_LOCK_ CLOCKS.(Core and Bus Agents)	Bus cycles when a LOCK signal asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: ▪ Uncacheable memory. ▪ Locked operation that spans two cache lines. ▪ Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses.
64H	See Table 18-61.	BUS_DATA_ RCV.(Core)	Bus cycles while processor receives data.	This event counts the number of bus cycles during which the processor is busy receiving data.
65H	See Table 18-61 and Table 18-62.	BUS_TRANS_BRD.(Core and Bus Agents)	Burst read bus transactions.	This event counts the number of burst read transactions including: ▪ L1 data cache read misses (and L1 data cache hardware prefetches). ▪ L2 hardware prefetches by the DPL and L2 streamer. ▪ IFU read misses of cacheable lines. It does not include RFO transactions.
66H	See Table 18-61 and Table 18-62.	BUS_TRANS_RFO.(Core and Bus Agents)	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. It also counts RFO bus transactions due to locked operations.
67H	See Table 18-61 and Table 18-62.	BUS_TRANS_WB. (Core and Bus Agents)	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-61 and Table 18-62.	BUS_TRANS_ IFETCH.(Core and Bus Agents)	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-61 and Table 18-62.	BUS_TRANS_ INVAL.(Core and Bus Agents)	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: ▪ A store operation hits a shared line in the L2 cache. ▪ A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-61 and Table 18-62.	BUS_TRANS_ PWR.(Core and Bus Agents)	Partial write bus transaction.	This event counts partial write bus transactions.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
6BH	See Table 18-61 and Table 18-62.	BUS_TRANS _P.(Core and Bus Agents)	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-61 and Table 18-62.	BUS_TRANS_IO.(Core and Bus Agents)	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-61 and Table 18-62.	BUS_TRANS_ DEF.(Core and Bus Agents)	Deferred bus transactions.	This event counts the number of deferred transactions.
6EH	See Table 18-61 and Table 18-62.	BUS_TRANS_ BURST.(Core and Bus Agents)	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: ▪ Burst reads. ▪ RFOs. ▪ Explicit writebacks. ▪ Write combine lines.
6FH	See Table 18-61 and Table 18-62.	BUS_TRANS_ MEM.(Core and Bus Agents)	Memory bus transactions.	This event counts all memory bus transactions including: ▪ Burst transactions. ▪ Partial reads and writes - invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_IVAL.
70H	See Table 18-61 and Table 18-62.	BUS_TRANS_ ANY.(Core and Bus Agents)	All bus transactions.	This event counts all bus transactions. This includes: ▪ Memory transactions. ▪ IO transactions (non memory-mapped). ▪ Deferred transaction completion. ▪ Other less frequent transactions, such as interrupts.
77H	See Table 18-61 and Table 18-65.	EXT_SNOOP. (Bus Agents, Snoop Response)	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. With the 'THIS_AGENT' mask, the event counts snoop responses from this processor to bus transactions sent by this processor. With the 'ALL_AGENTS' mask the event counts all snoop responses seen on the bus.
78H	See Table 18-61 and Table 18-66.	CMP_SNOOP.(Core, Snoop Type)	L1 data cache snooped by other core.	This event counts the number of times the L1 data cache is snooped for a cache line that is needed by the other core in the same processor. The cache line is either missing in the L1 instruction or data caches of the other core, or is available for reading only and the other core wishes to write the cache line.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				The snoop operation may change the cache line state. If the other core issued a read request that hit this core in E state, typically the state changes to S state in this core. If the other core issued a read for ownership request (due a write miss or hit to S state) that hits this core's cache line in E or S state, this typically results in invalidation of the cache line in this core. If the snoop hits a line in M state, the state is changed at a later opportunity. These snoops are performed through the L1 data cache store port. Therefore, frequent snoops may conflict with extensive stores to the L1 data cache, which may increase store latency and impact performance.
7AH	See Table 18-62.	BUS_HIT_DRV. (Bus Agents)	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response.
7BH	See Table 18-62.	BUS_HITM_DRV. (Bus Agents)	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response.
7DH	See Table 18-61.	BUSQ_EMPTY. (Core)	Bus queue empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. It also counts when the core is halted and the other core is not halted. This event can count occurrences for this core or both cores.
7EH	See Table 18-61 and Table 18-62.	SNOOP_STALL_ DRV.(Core and Bus Agents)	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. To obtain the number of bus cycles during which snoops on the bus are prohibited, multiply the event count by two. During the snoop stall cycles, no new bus transactions requiring a snoop response can be initiated on the bus. A bus agent asserts a snoop stall signal if it cannot response to a snoop request within three bus cycles.
7FH	See Table 18-61.	BUS_IO_WAIT. (Core)	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. With the SELF modifier this event counts IO requests per core. With the BOTH_CORE modifier, this event increments by one for any cycle for which there is a request from either core.
80H	00H	L1I_READS	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches that bypass the Instruction Fetch Unit (IFU).
81H	00H	L1I_MISSES	Instruction Fetch Unit misses.	This event counts all instruction fetches that miss the Instruction Fetch Unit (IFU) or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	02H	ITLB.SMALL_MISS	ITLB small page misses.	This event counts the number of instruction fetches from small pages that miss the ITLB.
82H	10H	ITLB.LARGE_MISS	ITLB large page misses.	This event counts the number of instruction fetches from large pages that miss the ITLB.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
82H	40H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes. This usually happens upon CR3 or CR0 writes, which are executed by the operating system during process switches.
82H	12H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches from either small or large pages that miss the ITLB.
83H	02H	INST_QUEUE.FULL	Cycles during which the instruction queue is full.	This event counts the number of cycles during which the instruction queue is full. In this situation, the core front end stops fetching more instructions. This is an indication of very long stalls in the back-end pipeline stages.
86H	00H	CYCLES_L1I_ MEM_STALLED	Cycles during which instruction fetches stalled.	This event counts the number of cycles for which an instruction fetch stalls, including stalls due to any of the following reasons: ▪ Instruction Fetch Unit cache misses. ▪ Instruction TLB misses. ▪ Instruction TLB faults.
87H	00H	ILD_STALL	Instruction Length Decoder stall cycles due to a length changing prefix.	This event counts the number of cycles during which the instruction length decoder uses the slow length decoder. Usually, instruction length decoding is done in one cycle. When the slow decoder is used, instruction decoding requires 6 cycles. The slow decoder is used in the following cases: ▪ Operand override prefix (66H) preceding an instruction with immediate data. ▪ Address override prefix (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. To avoid instruction length decoding stalls, generate code using imm8 or imm32 values instead of imm16 values. If you must use an imm16 value, store the value in a register using "mov reg, imm32" and use the register format of the instruction.
88H	00H	BR_INST_EXEC	Branch instructions executed.	This event counts all executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.
89H	00H	BR_MISSP_EXEC	Mispredicted branch instructions executed.	This event counts the number of mispredicted branch instructions that were executed.
8AH	00H	BR_BAC_ MISSP_EXEC	Branch instructions mispredicted at decoding.	This event counts the number of branch instructions that were mispredicted at decoding.
8BH	00H	BR_CND_EXEC	Conditional branch instructions executed.	This event counts the number of conditional branch instructions executed, but not necessarily retired.
8CH	00H	BR_CND_ MISSP_EXEC	Mispredicted conditional branch instructions executed.	This event counts the number of mispredicted conditional branch instructions that were executed.
8DH	00H	BR_IND_EXEC	Indirect branch instructions executed.	This event counts the number of indirect branch instructions that were executed.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
8EH	00H	BR_IND_MISSP _EXEC	Mispredicted indirect branch instructions executed.	This event counts the number of mispredicted indirect branch instructions that were executed.
8FH	00H	BR_RET_EXEC	RET instructions executed.	This event counts the number of RET instructions that were executed.
90H	00H	BR_RET_ MISSP_EXEC	Mispredicted RET instructions executed.	This event counts the number of mispredicted RET instructions that were executed.
91H	00H	BR_RET_BAC_ MISSP_EXEC	RET instructions executed mispredicted at decoding.	This event counts the number of RET instructions that were executed and were mispredicted at decoding.
92H	00H	BR_CALL_EXEC	CALL instructions executed.	This event counts the number of CALL instructions executed.
93H	00H	BR_CALL_ MISSP_EXEC	Mispredicted CALL instructions executed.	This event counts the number of mispredicted CALL instructions that were executed.
94H	00H	BR_IND_CALL_ EXEC	Indirect CALL instructions executed.	This event counts the number of indirect CALL instructions that were executed.
97H	00H	BR_TKN_ BUBBLE_1	Branch predicted taken with bubble 1.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
98H	00H	BR_TKN_ BUBBLE_2	Branch predicted taken with bubble 2.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
A0H	00H	RS_UOPS_ DISPATCHED	Micro-ops dispatched for execution.	This event counts the number of micro-ops dispatched for execution. Up to six micro-ops can be dispatched in each cycle.
A1H	01H	RS_UOPS_ DISPATCHED.PORT0	Cycles micro-ops dispatched for execution on port 0.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Issue Ports are described in *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Use IA32_PMC0 only.
A1H	02H	RS_UOPS_ DISPATCHED.PORT1	Cycles micro-ops dispatched for execution on port 1.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	04H	RS_UOPS_ DISPATCHED.PORT2	Cycles micro-ops dispatched for execution on port 2.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
A1H	08H	RS_UOPS_ DISPATCHED.PORT3	Cycles micro-ops dispatched for execution on port 3.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	10H	RS_UOPS_ DISPATCHED.PORT4	Cycles micro-ops dispatched for execution on port 4.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	20H	RS_UOPS_ DISPATCHED.PORT5	Cycles micro-ops dispatched for execution on port 5.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
AAH	01H	MACRO_INSTS. DECODED	Instructions decoded.	This event counts the number of instructions decoded (but not necessarily executed or retired).
AAH	08H	MACRO_INSTS. CISC_DECODED	CISC Instructions decoded.	This event counts the number of complex instructions decoded. Complex instructions usually have more than four micro-ops. Only one complex instruction can be decoded at a time.
ABH	01H	ESP.SYNCH	ESP register content synchron-ization.	This event counts the number of times that the ESP register is explicitly used in the address expression of a load or store operation, after it is implicitly used, for example by a push or a pop instruction. ESP synch micro-op uses resources from the rename pipe-stage and up to retirement. The expected ratio of this event divided by the number of ESP implicit changes is 0,2. If the ratio is higher, consider rearranging your code to avoid ESP synchronization events.
ABH	02H	ESP.ADDITIONS	ESP register automatic additions.	This event counts the number of ESP additions performed automatically by the decoder. A high count of this event is good, since each automatic addition performed by the decoder saves a micro-op from the execution units. To maximize the number of ESP additions performed automatically by the decoder, choose instructions that implicitly use the ESP, such as PUSH, POP, CALL, and RET instructions whenever possible.
B0H	00H	SIMD_UOPS_EXEC	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. It does not count MOVQ and MOVD stores from register to memory.
B1H	00H	SIMD_SAT_UOP_ EXEC	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B3H	01H	SIMD_UOP_ TYPE_EXEC.MUL	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHI FT	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	04H	SIMD_UOP_TYPE_EXEC.PA CK	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	08H	SIMD_UOP_TYPE_EXEC.UN PACK	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	10H	SIMD_UOP_TYPE_EXEC.LO GICAL	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers. INST_RETIRED.ANY_P is an architectural performance event.
C0H	01H	INST_RETIRED.LOADS	Instructions retired, which contain a load.	This event counts the number of instructions retired that contain a load operation.
C0H	02H	INST_RETIRED.STORES	Instructions retired, which contain a store.	This event counts the number of instructions retired that contain a store operation.
C0H	04H	INST_RETIRED.OTHER	Instructions retired, with no load or store operation.	This event counts the number of instructions retired that do not contain a load or a store operation.
C1H	01H	X87_OPS_RETIRED.FXCH	FXCH instructions retired.	This event counts the number of FXCH instructions retired. Modern compilers generate more efficient code and are less likely to use this instruction. If you obtain a high count for this event consider recompiling the code.
C1H	FEH	X87_OPS_RETIRED.ANY	Retired floating-point computational operations (precise event).	This event counts the number of floating-point computational operations retired. It counts: ▪ Floating point computational operations executed by the assist handler. ▪ Sub-operations of complex floating-point instructions like transcendental instructions. This event does not count: ▪ Floating-point computational operations that cause traps or assists. ▪ Floating-point loads and stores. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C2H	01H	UOPS_RETIRED.LD_IND_BR	Fused load+op or load+indirect branch retired.	This event counts the number of retired micro-ops that fused a load with another operation. This includes: ▪ Fusion of a load and an arithmetic operation, such as with the following instruction: ADD EAX, [EBX] where the content of the memory location specified by EBX register is loaded, added to EXA register, and the result is stored in EAX. ▪ Fusion of a load and a branch in an indirect branch operation, such as with the following instructions: ▪ JMP [RDI+200] ▪ RET ▪ Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C2H	02H	UOPS_RETIRED. STD_STA	Fused store address + data retired.	This event counts the number of store address calculations that are fused with store data emission into one micro-op. Traditionally, each store operation required two micro-ops. This event counts fusion of retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	04H	UOPS_RETIRED. MACRO_FUSION	Retired instruction pairs fused into one micro-op.	This event counts the number of times CMP or TEST instructions were fused with a conditional branch instruction into one micro-op. It counts fusion by retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code uses the processor resources more effectively.
C2H	07H	UOPS_RETIRED. FUSED	Fused micro-ops retired.	This event counts the total number of retired fused micro-ops. The counts include the following fusion types: ▪ Fusion of load operation with an arithmetic operation or with an indirect branch (counted by event UOPS_RETIRED.LD_IND_BR) ▪ Fusion of store address and data (counted by event UOPS_RETIRED.STD_STA) ▪ Fusion of CMP or TEST instruction with a conditional branch instruction (counted by event UOPS_RETIRED.MACRO_FUSION) Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	08H	UOPS_RETIRED. NON_FUSED	Non-fused micro-ops retired.	This event counts the number of micro-ops retired that were not fused.
C2H	0FH	UOPS_RETIRED. ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_ NUKES.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C3H	04H	MACHINE_NUKES.MEM_ORDER	Execution pipeline restart due to memory ordering conflict or memory disambiguation misprediction.	This event counts the number of times the pipeline is restarted due to either multi-threaded memory ordering conflicts or memory disambiguation misprediction. A multi-threaded memory ordering conflict occurs when a store, which is executed in another core, hits a load that is executed out of order in this core but not yet retired. As a result, the load needs to be restarted to satisfy the memory ordering model. See Chapter 8, "Multiple-Processor Management" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. To count memory disambiguation mispredictions, use the event MEMORY_DISAMBIGUATION.RESET.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. This is an architectural performance event.
C4H	01H	BR_INST_RETIRED.PRED_NOT_TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRED_NOT_TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions. (precise event)	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. This is an architectural performance event.
C6H	01H	CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired SSE packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired SSE scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired SSE2 packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired SSE2 scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C7H	10H	SIMD_INST_ RETIRED.VECTOR	Retired SSE2 vector integer instructions.	This event counts the number of SSE2 vector integer instructions retired.
C7H	1FH	SIMD_INST_ RETIRED.ANY	Retired Streaming SIMD instructions (precise event).	This event counts the overall number of retired SIMD instructions that use XMM registers. To count each type of SIMD instruction separately, use the following events: ▪ SIMD_INST_RETIRED.PACKED_SINGLE ▪ SIMD_INST_RETIRED.SCALAR_SINGLE ▪ SIMD_INST_RETIRED.PACKED_DOUBLE ▪ SIMD_INST_RETIRED.SCALAR_DOUBLE ▪ and SIMD_INST_RETIRED.VECTOR When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor.
C9H	00H	ITLB_MISS_ RETIRED	Retired instructions that missed the ITLB.	This event counts the number of retired instructions that missed the ITLB when they were fetched.
CAH	01H	SIMD_COMP_ INST_RETIRED. PACKED_SINGLE	Retired computational SSE packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_ INST_RETIRED. SCALAR_SINGLE	Retired computational SSE scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_ INST_RETIRED. PACKED_DOUBLE	Retired computational SSE2 packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	08H	SIMD_COMP_INST_RETIRE D.SCALAR_DOUBLE	Retired computational SSE2 scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CBH	01H	MEM_LOAD_ RETIRED.L1D _MISS	Retired loads that miss the L1 data cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache. This includes loads from cache lines that are currently being fetched, due to a previous L1 data cache miss to the same cache line. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	02H	MEM_LOAD_ RETIRED.L1D_ LINE_MISS	L1 data cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L1 data cache and send a request to the L2 cache to fetch the missing cache line. That is the missing cache line fetching has not yet started. The event count is equal to the number of cache lines fetched from the L2 cache by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events). When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	04H	MEM_LOAD_ RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache. This event counts loads from cacheable memory only. It does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CBH	08H	MEM_LOAD_ RETIRED.L2_LINE_MISS	L2 cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L2 cache and result in a bus request to fetch the missing cache line. That is the missing cache line fetching has not yet started. This event count is equal to the number of cache lines fetched from memory by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events). When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	10H	MEM_LOAD_ RETIRED.DTLB_ MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CCH	01H	FP_MMX_TRANS_TO_MMX	Transitions from Floating Point to MMX Instructions.	This event counts the first MMX instructions following a floating-point instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.
CCH	02H	FP_MMX_TRANS_TO_FP	Transitions from MMX Instructions to Floating Point Instructions.	This event counts the first floating-point instructions following any MMX instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed, changing the MMX state in the floating point stack.
CEH	00H	SIMD_INSTR_ RETIRED	SIMD Instructions retired.	This event counts the number of retired SIMD instructions that use MMX registers.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
D2H	01H	RAT_STALLS. ROB_READ_PORT	ROB read port stalls cycles.	This event counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read-port stall is counted again.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
D2H	02H	RAT_STALLS. PARTIAL_CYCLES	Partial register stall cycles.	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction uses a register that was partially written by previous instructions.
D2H	04H	RAT_STALLS. FLAGS	Flag stall cycles.	This event counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: ▪ An instruction modifies some, but not all, of the flags in the flag register. ▪ The next instruction, which depends on flags, depends on flags that were not modified by this instruction.
D2H	08H	RAT_STALLS. FPSW	FPU status word stall.	This event indicates that the FPU status word (FPSW) is written. To obtain the number of times the FPSW is written divide the event count by 2. The FPSW is written by instructions with long latency; a small count may indicate a high penalty.
D2H	0FH	RAT_STALLS. ANY	All RAT stall cycles.	This event counts the number of stall cycles due to conditions described by: ▪ RAT_STALLS.ROB_READ_PORT ▪ RAT_STALLS.PARTIAL ▪ RAT_STALLS.FLAGS ▪ RAT_STALLS.FPSW.
D4H	01H	SEG_RENAME_ STALLS.ES	Segment rename stalls - ES.	This event counts the number of stalls due to the lack of renaming resources for the ES segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	02H	SEG_RENAME_ STALLS.DS	Segment rename stalls - DS.	This event counts the number of stalls due to the lack of renaming resources for the DS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	04H	SEG_RENAME_ STALLS.FS	Segment rename stalls - FS.	This event counts the number of stalls due to the lack of renaming resources for the FS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	08H	SEG_RENAME_ STALLS.GS	Segment rename stalls - GS.	This event counts the number of stalls due to the lack of renaming resources for the GS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	0FH	SEG_RENAME_ STALLS.ANY	Any (ES/DS/FS/GS) segment rename stall.	This event counts the number of stalls due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
D5H	01H	SEG_REG_ RENAMES.ES	Segment renames - ES.	This event counts the number of times the ES segment register is renamed.
D5H	02H	SEG_REG_ RENAMES.DS	Segment renames - DS.	This event counts the number of times the DS segment register is renamed.
D5H	04H	SEG_REG_ RENAMES.FS	Segment renames - FS.	This event counts the number of times the FS segment register is renamed.
D5H	08H	SEG_REG_ RENAMES.GS	Segment renames - GS.	This event counts the number of times the GS segment register is renamed.
D5H	0FH	SEG_REG_ RENAMES.ANY	Any (ES/DS/FS/GS) segment rename.	This event counts the number of times any of the four segment registers (ES/DS/FS/GS) is renamed.
DCH	01H	RESOURCE_ STALLS.ROB_FULL	Cycles during which the ROB full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit the processor can handle. A high count for this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.
DCH	02H	RESOURCE_ STALLS.RS_FULL	Cycles during which the RS full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.
DCH	04	RESOURCE_ STALLS.LD_ST	Cycles during which the pipeline has exceeded load or store limit or waiting to commit all stores.	This event counts the number of cycles while resource-related stalls occur due to: ▪ The number of load instructions in the pipeline reached the limit the processor can handle. The stall ends when a loading instruction retires. ▪ The number of store instructions in the pipeline reached the limit the processor can handle. The stall ends when a storing instruction commits its data to the cache or memory. ▪ There is an instruction in the pipe that can be executed only when all previous stores complete and their data is committed in the caches or memory. For example, the SFENCE and MFENCE instructions require this behavior.
DCH	08H	RESOURCE_ STALLS.FPCW	Cycles stalled due to FPU control word write.	This event counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.
DCH	10H	RESOURCE_ STALLS.BR_MISS_CLEAR	Cycles stalled due to branch misprediction.	This event counts the number of cycles after a branch misprediction is detected at execution until the branch and all older micro-ops retire. During this time new micro-ops cannot enter the out-of-order pipeline.

Table 19-25. Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
DCH	1FH	RESOURCE_ STALLS.ANY	Resource related stalls.	This event counts the number of cycles while resource-related stalls occurs for any conditions described by the following events: ▪ RESOURCE_STALLS.ROB_FULL ▪ RESOURCE_STALLS.RS_FULL ▪ RESOURCE_STALLS.LD_ST ▪ RESOURCE_STALLS.FPCW ▪ RESOURCE_STALLS.BR_MISS_CLEAR
E0H	00H	BR_INST_ DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	00H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event. This occurs mainly after task switches.
E6H	00H	BACLEARS	BACLEARS asserted.	This event counts the number of times the front end is resteered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front and. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted costs approximately 7 cycles of instruction fetch. The effect on total execution time depends on the surrounding code.
F0H	00H	PREF_RQSTS_UP	Upward prefetches issued from DPL.	This event counts the number of upward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.
F8H	00H	PREF_RQSTS_DN	Downward prefetches issued from DPL.	This event counts the number of downward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.

19.13 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE GOLDMONT PLUS MICROARCHITECTURE

Intel Atom processors based on the Goldmont Plus microarchitecture support the architectural performance monitoring events listed in Table 19-1 and fixed-function performance events using a fixed counter. They also support the following performance monitoring events listed in Table 19-27. These events apply to processors with CPUID signature of 06_7AH. In addition, processors based on the Goldmont Plus microarchitecture also support the events listed in Table 19-27 (see Section 19.14, "Performance Monitoring Events for Processors Based on the Goldmont Microarchitecture"). For an event listed in Table 19-27 that also appears in the model-specific tables of prior generations, Table 19-27 supersedes prior generation tables.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

In Goldmont Plus microarchitecture, performance monitoring events that support Processor Event Based Sampling (PEBS) and PEBS records that contain processor state information that are associated with at-retirement tagging are marked by "Precise Event".

Table 19-26. Performance Events for the Goldmont Plus Microarchitecture

Event Num.	Umask Value	Event Name	Description	Comment
00H	01H	INST_RETIRED.ANY	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 inside interrupt handlers. This event uses fixed counter 0. You cannot collect a PEBS record for this event.	Fixed Event, Precise Event, Not Reduced Skid
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Counts page walks completed due to demand data loads (including SW prefetches) whose address translations missed in all TLB levels and were mapped to 4K pages. The page walks can end with or without a page fault.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Counts page walks completed due to demand data loads (including SW prefetches) whose address translations missed in all TLB levels and were mapped to 2M or 4M pages. The page walks can end with or without a page fault.	
08H	08H	DTLB_LOAD_MISSES.WALK_COMPLETED_1GB	Counts page walks completed due to demand data loads (including SW prefetches) whose address translations missed in all TLB levels and were mapped to 1GB pages. The page walks can end with or without a page fault.	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts once per cycle for each page walk occurring due to a load (demand data loads or SW prefetches). Includes cycles spent traversing the Extended Page Table (EPT). Average cycles per walk can be calculated by dividing by the number of walks.	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Counts page walks completed due to demand data stores whose address translations missed in the TLB and were mapped to 4K pages. The page walks can end with or without a page fault.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Counts page walks completed due to demand data stores whose address translations missed in the TLB and were mapped to 2M or 4M pages. The page walks can end with or without a page fault.	
49H	08H	DTLB_STORE_MISSES.WALK_COMPLETED_1GB	Counts page walks completed due to demand data stores whose address translations missed in the TLB and were mapped to 1GB pages. The page walks can end with or without a page fault.	
49H	10H	DTLB_STORE_MISSES.WALK_PENDING	Counts once per cycle for each page walk occurring due to a demand data store. Includes cycles spent traversing the Extended Page Table (EPT). Average cycles per walk can be calculated by dividing by the number of walks.	

Table 19-26. Performance Events for the Goldmont Plus Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
4FH	10H	EPT.WALK_PENDING	Counts once per cycle for each page walk only while traversing the Extended Page Table (EPT), and does not count during the rest of the translation. The EPT is used for translating Guest-Physical Addresses to Physical Addresses for Virtual Machine Monitors (VMMs). Average cycles per walk can be calculated by dividing the count by number of walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Counts page walks completed due to instruction fetches whose address translations missed in the TLB and were mapped to 4K pages. The page walks can end with or without a page fault.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Counts page walks completed due to instruction fetches whose address translations missed in the TLB and were mapped to 2M or 4M pages. The page walks can end with or without a page fault.	
85H	08H	ITLB_MISSES.WALK_COMPLETED_1GB	Counts page walks completed due to instruction fetches whose address translations missed in the TLB and were mapped to 1GB pages. The page walks can end with or without a page fault.	
85H	10H	ITLB_MISSES.WALK_PENDING	Counts once per cycle for each page walk occurring due to an instruction fetch. Includes cycles spent traversing the Extended Page Table (EPT). Average cycles per walk can be calculated by dividing by the number of walks.	
BDH	20H	TLB_FLUSHES.STLB_ANY	Counts STLB flushes. The TLBs are flushed on instructions like INVLPG and MOV to CR3.	
C3H	20H	MACHINE_CLEARS.PAGE_FAULT	Counts the number of times that the machines clears due to a page fault. Covers both I-side and D-side (Loads/Stores) page faults. A page fault occurs when either page is not present, or an access violation.	

19.14 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE GOLDMONT MICROARCHITECTURE

Intel Atom processors based on the Goldmont microarchitecture support the architectural performance monitoring events listed in Table 19-1 and fixed-function performance events using a fixed counter. In addition, they also support the following model-specific performance monitoring events listed in Table 19-27. These events apply to processors with CPUID signatures of 06_5CH, 06_5FH, and 06_7AH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

In Goldmont microarchitecture, performance monitoring events that support Processor Event Based Sampling (PEBS) and PEBS records that contain processor state information that are associated with at-retirement tagging are marked by "Precise Event".

Table 19-27. Performance Events for the Goldmont Microarchitecture

Event Num.	Umask Value	Event Name	Description	Comment
03H	10H	LD_BLOCKS.ALL_BLOCK	Counts anytime a load that retires is blocked for any reason.	Precise Event
03H	08H	LD_BLOCKS.UTLB_MISS	Counts loads blocked because they are unable to find their physical address in the micro TLB (UTLB).	Precise Event
03H	02H	LD_BLOCKS.STORE_FORWARD	Counts a load blocked from using a store forward because of an address/size mismatch; only one of the loads blocked from each store will be counted.	Precise Event

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Counts a load blocked from using a store forward, but did not occur because the store data was not available at the right time. The forward might occur subsequently when the data is available.	Precise Event
03H	04H	LD_BLOCKS.4K_ALIAS	Counts loads that block because their address modulo 4K matches a pending store.	Precise Event
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts every core cycle when a Data-side (walks due to data operation) page walk is in progress.	
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts every core cycle when an Instruction-side (walks due to an instruction fetch) page walk is in progress.	
05H	03H	PAGE_WALKS.CYCLES	Counts every core cycle a page-walk is in progress due to either a data memory operation, or an instruction fetch.	
0EH	00H	UOPS_ISSUED.ANY	Counts uops issued by the front end and allocated into the back end of the machine. This event counts uops that retire as well as uops that were speculatively executed but didn't retire. The sort of speculative uops that might be counted includes, but is not limited to those uops issued in the shadow of a mispredicted branch, those uops that are inserted during an assist (such as for a denormal floating-point result), and (previously allocated) uops that might be canceled during a machine clear.	
13H	02H	MISALIGN_MEM_REF.LOAD_PAGE_SPLIT	Counts when a memory load of a uop that spans a page boundary (a split) is retired.	Precise Event
13H	04H	MISALIGN_MEM_REF.STORE_PAGE_SPLIT	Counts when a memory store of a uop that spans a page boundary (a split) is retired.	Precise Event
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts memory requests originating from the core that reference a cache line in the L2 cache.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts memory requests originating from the core that miss in the L2 cache.	
30H	00H	L2_REJECT_XQ.ALL	Counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the intra-die interconnect (IDI) fabric. The XQ may reject transactions from the L2Q (non-cacheable requests), L2 misses and L2 write-back victims.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to ensure fairness between cores, or to delay a core's dirty eviction when the address conflicts with incoming external snoops.	
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted. This event uses a programmable general purpose performance counter.	
3CH	01H	CPU_CLK_UNHALTED.REF	Reference cycles when core is not halted. This event uses a programmable general purpose performance counter.	
51H	01H	DL1.DIRTY_EVICTION	Counts when a modified (dirty) cache line is evicted from the data L1 cache and needs to be written back to memory. No count will occur if the evicted line is clean, and hence does not require a writeback.	

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
80H	01H	ICACHE.HIT	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is in the Icache (hit). The event strives to count on a cache line basis, so that multiple accesses which hit in a single cache line count as one ICACHE.HIT. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	02H	ICACHE.MISSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is not in the Icache (miss). The event strives to count on a cache line basis, so that multiple accesses which miss in a single cache line count as one ICACHE.MISS. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is not in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	03H	ICACHE.ACCESSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line. The event strives to count on a cache line basis, so that multiple fetches to a single cache line count as one ICACHE.ACCESS. Specifically, the event counts when accesses from straight line code crosses the cache line boundary, or when a branch target is to a new line. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
81H	04H	ITLB.MISS	Counts the number of times the machine was unable to find a translation in the Instruction Translation Lookaside Buffer (ITLB) for a linear address of an instruction fetch. It counts when new translations are filled into the ITLB. The event is speculative in nature, but will not count translations (page walks) that are begun and not finished, or translations that are finished but not filled into the ITLB.	
86H	00H	FETCH_STALL.ALL	Counts cycles that fetch is stalled due to any reason. That is, the decoder queue is able to accept bytes, but the fetch unit is unable to provide bytes. This will include cycles due to an ITLB miss, ICache miss and other events.	
86H	01H	FETCH_STALL.ITLB_FILL_PENDING_CYCLES	Counts cycles that fetch is stalled due to an outstanding ITLB miss. That is, the decoder queue is able to accept bytes, but the fetch unit is unable to provide bytes due to an ITLB miss. Note: this event is not the same as page walk cycles to retrieve an instruction translation.	
86H	02H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts cycles that an ICache miss is outstanding, and instruction fetch is stalled. That is, the decoder queue is able to accept bytes, but the fetch unit is unable to provide bytes, while an Icache miss is outstanding. Note this event is not the same as cycles to retrieve an instruction due to an Icache miss. Rather, it is the part of the Instruction Cache (ICache) miss time where no bytes are available for the decoder.	

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
9CH	00H	UOPS_NOT_DELIVERED.ANY	This event is used to measure front-end inefficiencies, i.e., when the front end of the machine is not delivering uops to the back end and the back end has not stalled. This event can be used to identify if the machine is truly front-end bound. When this event occurs, it is an indication that the front end of the machine is operating at less than its theoretical peak performance. Background: We can think of the processor pipeline as being divided into 2 broader parts: the front end and the back end. The front end is responsible for fetching the instruction, decoding into uops in machine understandable format and putting them into a uop queue to be consumed by the back end. The back end then takes these uops and allocates the required resources. When all resources are ready, uops are executed. If the back end is not ready to accept uops from the front end, then we do not want to count these as front-end bottlenecks. However, whenever we have bottlenecks in the back end, we will have allocation unit stalls and eventually force the front end to wait until the back end is ready to receive more uops. This event counts only when the back end is requesting more micro-uops and the front end is not able to provide them. When 3 uops are requested and no uops are delivered, the event counts 3. When 3 are requested, and only 1 is delivered, the event counts 2. When only 2 are delivered, the event counts 1. Alternatively stated, the event will not count if 3 uops are delivered, or if the back end is stalled and not requesting any uops at all. Counts indicate missed opportunities for the front end to deliver a uop to the back end. Some examples of conditions that cause front-end efficiencies are: Icache misses, ITLB misses, and decoder restrictions that limit the front-end bandwidth. Known Issues: Some uops require multiple allocation slots. These uops will not be charged as a front end 'not delivered' opportunity, and will be regarded as a back-end problem. For example, the INC instruction has one uop that requires 2 issue slots. A stream of INC instructions will not count as UOPS_NOT_DELIVERED, even though only one instruction can be issued per clock. The low uop issue rate for a stream of INC instructions is considered to be a back-end issue.	
B7H	01H, 02H	OFFCORE_RESPONSE	Requires MSR_OFFCORE_RESP[0,1] to specify request type and response. (Duplicated for both MSRs.)	
C0H	00H	INST_RETIRED.ANY_P	Counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The event continues counting during hardware interrupts, traps, and inside interrupt handlers. This is an architectural performance event. This event uses a programmable general purpose performance counter. *This event is a Precise Event: the EventingRIP field in the PEBS record is precise to the address of the instruction which caused the event. Note: Because PEBS records can be collected only on IA32_PMC0, only one event can use the PEBS facility at a time.	Precise Event
C2H	00H	UOPS_RETIRED.ANY	Counts uops which have retired.	Precise Event, Not Reduced Skid

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C2H	01H	UOPS_RETIRED.MS	Counts uops retired that are from the complex flows issued by the micro-sequencer (MS). Counts both the uops from a micro-coded instruction, and the uops that might be generated from a micro-coded assist.	Precise Event, Not Reduced Skid
C2H	08H	UOPS_RETIRED.FPDIV	Counts the number of floating point divide uops retired.	Precise Event
C2H	10H	UOPS_RETIRED.IDIV	Counts the number of integer divide uops retired.	Precise Event
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the processor detects that a program is writing to a code section and has to perform a machine clear because of that modification. Self-modifying code (SMC) causes a severe penalty in all Intel architecture processors.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts machine clears due to memory ordering issues. This occurs when a snoop request happens and the machine is uncertain if memory ordering will be preserved as another core is in the process of modifying the data.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts machine clears due to floating-point (FP) operations needing assists. For instance, if the result was a floating-point denormal, the hardware clears the pipeline and reissues uops to produce the correct IEEE compliant denormal result.	
C3H	08H	MACHINE_CLEARS.DISAMBIGUATION	Counts machine clears due to memory disambiguation. Memory disambiguation happens when a load which has been issued conflicts with a previous un-retired store in the pipeline whose address was not known at issue time, but is later resolved to be the same as the load address.	
C3H	00H	MACHINE_CLEARS.ALL	Counts machine clears for any reason.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts branch instructions retired for all branch types. This is an architectural performance event.	Precise Event
C4H	7EH	BR_INST_RETIRED.JCC	Counts retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was taken and when it was not taken.	Precise Event
C4H	80H	BR_INST_RETIRED.ALL_TAKEN_BRANCHES	Counts the number of taken branch instructions retired.	Precise Event
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were taken and does not count when the Jcc branch instruction were not taken.	Precise Event
C4H	F9H	BR_INST_RETIRED.CALL	Counts near CALL branch instructions retired.	Precise Event
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts near relative CALL branch instructions retired.	Precise Event
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts near indirect CALL branch instructions retired.	Precise Event
C4H	F7H	BR_INST_RETIRED.RETURN	Counts near return branch instructions retired.	Precise Event
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts near indirect call or near indirect jmp branch instructions retired.	Precise Event
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts far branch instructions retired. This includes far jump, far call and return, and Interrupt call and return.	Precise Event
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts mispredicted branch instructions retired including all branch types.	Precise Event

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C5H	7EH	BR_MISP_RETIRED.JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was supposed to be taken and when it was not supposed to be taken (but the processor predicted the opposite condition).	Precise Event
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were supposed to be taken but the processor predicted that it would not be taken.	Precise Event
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts mispredicted near indirect CALL branch instructions retired, where the target address taken was not what the processor predicted.	Precise Event
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts mispredicted near RET branch instructions retired, where the return address taken was not what the processor predicted.	Precise Event
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts mispredicted branch instructions retired that were near indirect call or near indirect jmp, where the target address taken was not what the processor predicted.	Precise Event
CAH	01H	ISSUE_SLOTS_NOT_CONSUMED.RESOURCE_FULL	Counts the number of issue slots per core cycle that were not consumed because of a full resource in the back end. Including but not limited to resources include the Re-order Buffer (ROB), reservation stations (RS), load/store buffers, physical registers, or any other needed machine resource that is currently unavailable. Note that uops must be available for consumption in order for this event to fire. If a uop is not available (Instruction Queue is empty), this event will not count.	
CAH	02H	ISSUE_SLOTS_NOT_CONSUMED.RECOVERY	Counts the number of issue slots per core cycle that were not consumed by the back end because allocation is stalled waiting for a mispredicted jump to retire or other branch-like conditions (e.g. the event is relevant during certain microcode flows). Counts all issue slots blocked while within this window, including slots where uops were not available in the Instruction Queue.	
CAH	00H	ISSUE_SLOTS_NOT_CONSUMED.ANY	Counts the number of issue slots per core cycle that were not consumed by the back end due to either a full resource in the back end (RESOURCE_FULL), or due to the processor recovering from some event (RECOVERY).	
CBH	01H	HW_INTERRUPTS.RECEIVED	Counts hardware interrupts received by the processor.	
CBH	02H	HW_INTERRUPTS.MASKED	Counts the number of core cycles during which interrupts are masked (disabled). Increments by 1 each core cycle that EFLAGS.IF is 0, regardless of whether interrupts are pending or not.	
CBH	04H	HW_INTERRUPTS.PENDING_AND_MASKED	Counts core cycles during which there are pending interrupts, but interrupts are masked (EFLAGS.IF = 0).	
CDH	00H	CYCLES_DIV_BUSY.ALL	Counts core cycles if either divide unit is busy.	
CDH	01H	CYCLES_DIV_BUSY.IDIV	Counts core cycles if the integer divide unit is busy.	
CDH	02H	CYCLES_DIV_BUSY.FPDIV	Counts core cycles if the floating point divide unit is busy.	
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	Counts the number of load uops retired.	Precise Event
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	Counts the number of store uops retired.	Precise Event

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
D0H	83H	MEM_UOPS_RETIRED.ALL	Counts the number of memory uops retired that are either a load or a store or both.	Precise Event
D0H	11H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Counts load uops retired that caused a DTLB miss.	Precise Event
D0H	12H	MEM_UOPS_RETIRED.DTLB_MISS_STORES	Counts store uops retired that caused a DTLB miss.	Precise Event
D0H	13H	MEM_UOPS_RETIRED.DTLB_MISS	Counts uops retired that had a DTLB miss on load, store or either. Note that when two distinct memory operations to the same page miss the DTLB, only one of them will be recorded as a DTLB miss.	Precise Event
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Counts locked memory uops retired. This includes 'regular' locks and bus locks. To specifically count bus locks only, see the offcore response event. A locked access is one with a lock prefix, or an exchange to memory.	Precise Event
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Counts load uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Counts store uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	43H	MEM_UOPS_RETIRED.SPLIT	Counts memory uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Counts load uops retired that hit the L1 data cache.	Precise Event
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Counts load uops retired that miss the L1 data cache.	Precise Event
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Counts load uops retired that hit in the L2 cache.	Precise Event
0xD1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Counts load uops retired that miss in the L2 cache.	Precise Event
D1H	20H	MEM_LOAD_UOPS_RETIRED.HITM	Counts load uops retired where the cache line containing the data was in the modified state of another core or modules cache (HITM). More specifically, this means that when the load address was checked by other caching agents (typically another processor) in the system, one of those caching agents indicated that they had a dirty copy of the data. Loads that obtain a HITM response incur greater latency than most that is typical for a load. In addition, since HITM indicates that some other processor had this data in its cache, it implies that the data was shared between processors, or potentially was a lock or semaphore value. This event is useful for locating sharing, false sharing, and contended locks.	Precise Event

Table 19-27. Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
D1H	40H	MEM_LOAD_UOPS_RETIRED.WCB_HIT	Counts memory load uops retired where the data is retrieved from the WCB (or fill buffer), indicating that the load found its data while that data was in the process of being brought into the L1 cache. Typically a load will receive this indication when some other load or prefetch missed the L1 cache and was in the process of retrieving the cache line containing the data, but that process had not yet finished (and written the data back to the cache). For example, consider load X and Y, both referencing the same cache line that is not in the L1 cache. If load X misses cache first, it obtains and WCB (or fill buffer) begins the process of requesting the data. When load Y requests the data, it will either hit the WCB, or the L1 cache, depending on exactly what time the request to Y occurs.	Precise Event
D1H	80H	MEM_LOAD_UOPS_RETIRED.DRAM_HIT	Counts memory load uops retired where the data is retrieved from DRAM. Event is counted at retirement, so the speculative loads are ignored. A memory load can hit (or miss) the L1 cache, hit (or miss) the L2 cache, hit DRAM, hit in the WCB or receive a HITM response.	Precise Event
E6H	01H	BACLEARS.ALL	Counts the number of times a BACLEAR is signaled for any reason, including, but not limited to indirect branch/call, Jcc (Jump on Conditional Code/Jump if Condition is Met) branch, unconditional branch/call, and returns.	
E6H	08H	BACLEARS.RETURN	Counts BACLEARS on return instructions.	
E6H	10H	BACLEARS.COND	Counts BACLEARS on Jcc (Jump on Conditional Code/Jump if Condition is Met) branches.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the Microcode Sequencer (MS) starts a flow of uops from the MSROM. It does not count every time a uop is read from the MSROM. The most common case that this counts is when a micro-coded instruction is encountered by the front end of the machine. Other cases include when an instruction encounters a fault, trap, or microcode assist of any sort that initiates a flow of uops. The event will count MS startups for uops that are speculative, and subsequently cleared by branch mispredict or a machine clear.	
E9H	01H	DECODE_RESTRICTION.PREDECODE_WRONG	Counts the number of times the prediction (from the pre-decode cache) for instruction length is incorrect.	

19.15 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE SILVERMONT MICROARCHITECTURE

Processors based on the Silvermont microarchitecture support the architectural performance monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter. In addition, they also support the following model-specific performance monitoring events listed in Table 19-28. These processors have the CPUID signatures of 06_37H, 06_4AH, 06_4DH, 06_5AH, and 06_5DH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

Table 19-28. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
03H	01H	REHABQ.LD_BLOCK_ST_FORWARD	Loads blocked due to store forward restriction.	This event counts the number of retired loads that were prohibited from receiving forwarded data from the store because of address mismatch.
03H	02H	REHABQ.LD_BLOCK_STD_NOTREADY	Loads blocked due to store data not ready.	This event counts the cases where a forward was technically possible, but did not occur because the store data was not available at the right time.
03H	04H	REHABQ.ST_SPLITS	Store uops that split cache line boundary.	This event counts the number of retire stores that experienced cache line boundary splits.
03H	08H	REHABQ.LD_SPLITS	Load uops that split cache line boundary.	This event counts the number of retire loads that experienced cache line boundary splits.
03H	10H	REHABQ.LOCK	Uops with lock semantics.	This event counts the number of retired memory operations with lock semantics. These are either implicit locked instructions such as the XCHG instruction or instructions with an explicit LOCK prefix (F0H).
03H	20H	REHABQ.STA_FULL	Store address buffer full.	This event counts the number of retired stores that are delayed because there is not a store address buffer available.
03H	40H	REHABQ.ANY_LD	Any reissued load uops.	This event counts the number of load uops reissued from Rehabq.
03H	80H	REHABQ.ANY_ST	Any reissued store uops.	This event counts the number of store uops reissued from Rehabq.
04H	01H	MEM_UOPS_RETIRED.L1_MISS_LOADS	Loads retired that missed L1 data cache.	This event counts the number of load ops retired that miss in L1 Data cache. Note that prefetch misses will not be counted.
04H	02H	MEM_UOPS_RETIRED.L2_HIT_LOADS	Loads retired that hit L2.	This event counts the number of load micro-ops retired that hit L2.
04H	04H	MEM_UOPS_RETIRED.L2_MISS_LOADS	Loads retired that missed L2.	This event counts the number of load micro-ops retired that missed L2.
04H	08H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Loads missed DTLB.	This event counts the number of load ops retired that had DTLB miss.
04H	10H	MEM_UOPS_RETIRED.UTLB_MISS	Loads missed UTLB.	This event counts the number of load ops retired that had UTLB miss.
04H	20H	MEM_UOPS_RETIRED.HITM	Cross core or cross module hitm.	This event counts the number of load ops retired that got data from the other core or from the other module.
04H	40H	MEM_UOPS_RETIRED.ALL_LOADS	All Loads.	This event counts the number of load ops retired.
04H	80H	MEM_UOP_RETIRED.ALL_STORES	All Stores.	This event counts the number of store ops retired.
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Duration of D-side page-walks in core cycles.	This event counts every cycle when a D-side (walks due to a load) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Duration of I-side page-walks in core cycles.	This event counts every cycle when an I-side (walks due to an instruction fetch) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.

Table 19-28. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
05H	03H	PAGE_WALKS.WALKS	Total number of page-walks that are completed (I-side and D-side).	This event counts when a data (D) page walk or an instruction (I) page walk is completed or started. Since a page walk implies a TLB miss, the number of TLB misses can be counted by counting the number of pagewalks. Edge trigger bit must be set. Clear Edge to count the number of cycles.
2EH	41H	LONGEST_LAT_CACHE.MISS	L2 cache request misses.	This event counts the total number of L2 cache references and the number of L2 cache misses respectively. L3 is not supported in Silvermont microarchitecture.
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	L2 cache requests from this core.	This event counts requests originating from the core that references a cache line in the L2 cache. L3 is not supported in Silvermont microarchitecture.
30H	00H	L2_REJECT_XQ.ALL	Counts the number of request from the L2 that were not accepted into the XQ.	This event counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the IDI link. The XQ may reject transactions from the L2Q (non-cacheable requests), BBS (L2 misses) and WOB (L2 write-back victims).
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of request that were not accepted into the L2Q because the L2Q is FULL.	This event counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to insure fairness between cores, or to delay a core's dirty eviction when the address conflicts incoming external snoops. (Note that L2 prefetcher requests that are dropped are not counted by this event.).
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time.
N/A	N/A	CPU_CLK_UNHALTED.CORE	Core cycles when core is not halted.	This uses the fixed counter 1 to count the same condition as CPU_CLK_UNHALTED.CORE_P does.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Bus cycles when core is not halted.	This event counts the number of bus cycles that the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time. This event is not affected by core frequency changes.
N/A	N/A	CPU_CLK_UNHALTED.REF_TSC	Reference cycles when core is not halted.	This event counts the number of reference cycles at a TSC rate that the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time. This event is not affected by core frequency changes.
80H	01H	ICACHE.HIT	Instruction fetches from Icache.	This event counts all instruction fetches from the instruction cache.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.

Table 19-28. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.5.2.2.	Requires MSR_OFFCORE_RESP0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.5.2.2.	Requires MSR_OFFCORE_RESP1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (PEBS supported with IA32_PMC0).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
N/A	N/A	INST_RETIRED.ANY	Instructions retired.	This uses the fixed counter 0 to count the same condition as INST_RETIRED.ANY_P does.
C2H	01H	UOPS_RETIRED.MS	MSROM micro-ops retired.	This event counts the number of micro-ops retired that were supplied from MSROM.
C2H	10H	UOPS_RETIRED.ALL	Micro-ops retired.	This event counts the number of micro-ops retired.
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Stalls due to Memory ordering.	This event counts the number of times that pipeline was cleared due to memory ordering issues.
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Stalls due to FP assists.	This event counts the number of times that pipeline stalled due to FP operations needing assists.
C3H	08H	MACHINE_CLEARS.ALL	Stalls due to any causes.	This event counts the number of times that pipeline stalled due to due to any causes (including SMC, MO, FP assist, etc.).
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Retired branch instructions.	This event counts the number of branch instructions retired.
C4H	7EH	BR_INST_RETIRED.JCC	Retired branch instructions that were conditional jumps.	This event counts the number of branch instructions retired that were conditional jumps.
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Retired far branch instructions.	This event counts the number of far branch instructions retired.
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Retired instructions of near indirect Jmp or call.	This event counts the number of branch instructions retired that were near indirect call or near indirect jmp.
C4H	F7H	BR_INST_RETIRED.RETURN	Retired near return instructions.	This event counts the number of near RET branch instructions retired.
C4H	F9H	BR_INST_RETIRED.CALL	Retired near call instructions.	This event counts the number of near CALL branch instructions retired.
C4H	FBH	BR_INST_RETIRED.IND_CALL	Retired near indirect call instructions.	This event counts the number of near indirect CALL branch instructions retired.
C4H	FDH	BR_INST_RETIRED.REL_CALL	Retired near relative call instructions.	This event counts the number of near relative CALL branch instructions retired.
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Retired conditional jumps that were taken.	This event counts the number of branch instructions retired that were conditional jumps and taken.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Retired mispredicted branch instructions.	This event counts the number of mispredicted branch instructions retired.

Table 19-28. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C5H	7EH	BR_MISP_RETIRED.JCC	Retired mispredicted conditional jumps.	This event counts the number of mispredicted branch instructions retired that were conditional jumps.
C5H	BFH	BR_MISP_RETIRED.FAR	Retired mispredicted far branch instructions.	This event counts the number of mispredicted far branch instructions retired.
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Retired mispredicted instructions of near indirect Jmp or call.	This event counts the number of mispredicted branch instructions retired that were near indirect call or near indirect jmp.
C5H	F7H	BR_MISP_RETIRED.RETURN	Retired mispredicted near return instructions.	This event counts the number of mispredicted near RET branch instructions retired.
C5H	F9H	BR_MISP_RETIRED.CALL	Retired mispredicted near call instructions.	This event counts the number of mispredicted near CALL branch instructions retired.
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Retired mispredicted near indirect call instructions.	This event counts the number of mispredicted near indirect CALL branch instructions retired.
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Retired mispredicted near relative call instructions	This event counts the number of mispredicted near relative CALL branch instructions retired.
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Retired mispredicted conditional jumps that were taken.	This event counts the number of mispredicted branch instructions retired that were conditional jumps and taken.
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of cycles when no uops are allocated and a RATstall is asserted.	Counts the number of cycles when no uops are allocated and a RATstall is asserted.
CAH	3FH	NO_ALLOC_CYCLES.ALL	Front end not delivering.	This event counts the number of cycles when the front end does not provide any instructions to be allocated for any reason.
CAH	50H	NO_ALLOC_CYCLES.NOT_DELIVERED	Front end not delivering back end not stalled.	This event counts the number of cycles when the front end does not provide any instructions to be allocated but the back end is not stalled.
CBH	01H	RS_FULL_STALL.MEC	MEC RS full.	This event counts the number of cycles the allocation pipe line stalled due to the RS for the MEC cluster is full.
CBH	1FH	RS_FULL_STALL.ALL	Any RS full.	This event counts the number of cycles that the allocation pipe line stalled due to any one of the RS is full.
CDH	01H	CYCLES_DIV_BUSY.ANY	Divider Busy.	This event counts the number of cycles the divider is busy.
E6H	01H	BACLEARS.ALL	BACLEARS asserted for any branch.	This event counts the number of baclears for any type of branch.
E6H	08H	BACLEARS.RETURN	BACLEARS asserted for return branch.	This event counts the number of baclears for return branches.

Table 19-28. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
E6H	10H	BACLEARS.COND	BACLEARS asserted for conditional branch.	This event counts the number of baclears for conditional branches.
E7H	01H	MS_DECODED.MS_ENTRY	MS Decode starts.	This event counts the number of times the MSROM starts a flow of UOPS.

19.15.1 Performance Monitoring Events for Processors Based on the Airmont Microarchitecture

Intel processors based on the Airmont microarchitecture support the same architectural and the model-specific performance monitoring events as processors based on the Silvermont microarchitecture. All of the events listed in Table 19-28 apply. These processors have the CPUID signatures that include 06_4CH.

19.16 PERFORMANCE MONITORING EVENTS FOR 45 NM AND 32 NM INTEL® ATOM™ PROCESSORS

45 nm and 32 nm processors based on the Intel® Atom™ microarchitecture support the architectural performance monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter listed in Table 19-24. In addition, they also support the following model-specific performance monitoring events listed in Table 19-29.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors

Event Num.	Umask Value	Event Name	Definition	Description and Comment
02H	81H	STORe_FORWARDS.GOOD	Good store forwards.	This event counts the number of times store data was forwarded directly to a load.
06H	00H	SEGMENT_REG_LOADS.ANY	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.
07H	01H	PREFETCH.PREFETCHT0	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	06H	PREFETCH.SW_L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
07H	08H	PREFETCH.PREFETCHNTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
08H	07H	DATA_TLB_MISSES.DTLB_MISS	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	05H	DATA_TLB_MISSES.DTLB_MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	09H	DATA_TLB_MISSES.L0_DTLB_MISS_LD	L0_DTLB misses due to load operations.	This event counts the number of L0_DTLB misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	06H	DATA_TLB_MISSES.DTLB_MISS_ST	DTLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses.
0CH	03H	PAGE_WALKS.WALKS	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. This can hint to whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be set.
0CH	03H	PAGE_WALKS.CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. This can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be cleared.
10H	01H	X87_COMP_OPS_EXE.ANY.S	Floating point computational micro-ops executed.	This event counts the number of x87 floating point computational micro-ops executed.
10H	81H	X87_COMP_OPS_EXE.ANY.AR	Floating point computational micro-ops retired.	This event counts the number of x87 floating point computational micro-ops retired.
11H	01H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
11H	81H	FP_ASSIST.AR	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.
12H	01H	MUL.S	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations.
12H	81H	MUL.AR	Multiply operations retired.	This event counts the number of multiply operations retired. This includes integer as well as floating point multiply operations.
13H	01H	DIV.S	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed.
13H	81H	DIV.AR	Divide operations retired.	This event counts the number of divide operations retired. This includes integer divides, floating point divides and square-root operations executed.
14H	01H	CYCLES_DIV_BUSY	Cycles the driver is busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE.
21H	See Table 18-61	L2_ADS	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. This event can count occurrences for this core or both cores.
22H	See Table 18-61	L2_DBUS_BUSY	Cycles the L2 cache data bus is busy.	This event counts core cycles during which the L2 cache data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. The count will increment by two for a full cache-line request.
24H	See Table 18-61 and Table 18-63	L2_LINES_IN	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. This event can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-61	L2_M_LINES_IN	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
26H	See Table 18-61 and Table 18-63	L2_LINES_OUT	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-61 and Table 18-63	L2_M_LINES_OUT	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a shared-state in one of the L1 data caches. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
28H	See Table 18-61 and Table 18-64	L2_IFETCH	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the ICache. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
29H	See Table 18-61, Table 18-63 and Table 18-64	L2_LD	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. This event can count occurrences for this core or both cores. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together or separately. - of accesses to cache lines at different MESI states.
2AH	See Table 18-61 and Table 18-64	L2_ST	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2BH	See Table 18-61 and Table 18-64	L2_LOCK	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2EH	See Table 18-61, Table 18-63 and Table 18-64	L2_RQSTS	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together, or separately. - of accesses to cache lines at different MESI states.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
2EH	41H	L2_RQSTS.SELF.DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF.DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
30H	See Table 18-61, Table 18-63 and Table 18-64	L2_REJECT_BUSQ	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: - The bus queue is full. - The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. - For this core or both cores. - Due to demand requests and L2 hardware prefetch requests together, or separately. - Of accesses to cache lines at different MESI states.
32H	See Table 18-61	L2_NO_REQ	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests are pending.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions.	This event counts the number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions that include a frequency change, either with or without VID change. This event is incremented only while the counting core is in C0 state. In situations where an EIST transition was caused by hardware as a result of CxE state transitions, those EIST transitions will also be registered in this event. Enhanced Intel Speedstep Technology transitions are commonly initiated by OS, but can be initiated by HW internally. For example: CxE states are C-states (C1,C2,C3...) which not only place the CPU into a sleep state by turning off the clock and other components, but also lower the voltage (which reduces the leakage power consumption). The same is true for thermal throttling transition which uses Enhanced Intel Speedstep Technology internally.
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond, and returns to normal when the temperature falls below the thermal trip threshold temperature.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time. In systems with a constant core frequency, this event can give you a measurement of the elapsed time while the core was not in halt state by dividing the event count by the core frequency. -This is an architectural performance event. - The event CPU_CLK_UNHALTED.CORE_P is counted by a programmable counter. - The event CPU_CLK_UNHALTED.CORE is counted by a designated fixed counter, leaving the two programmable counters available for other events.
3CH	01H	CPU_CLK_UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give you a measurement of the elapsed time while the core was not in the halt state, by dividing the event count by the bus frequency. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_UNHALTED.NO_OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted, and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	21H	L1D_CACHE.LD	L1 Cacheable Data Reads.	This event counts the number of data reads from cacheable memory.
40H	22H	L1D_CACHE.ST	L1 Cacheable Data Writes.	This event counts the number of data writes to cacheable memory.
60H	See Table 18-61 and Table 18-62.	BUS_REQUEST_OUTSTANDING	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
61H	See Table 18-62.	BUS_BNR_DRV	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
62H	See Table 18-62.	BUS_DRDY_CLOCKS	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. This event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
63H	See Table 18-61 and Table 18-62.	BUS_LOCK_CLOCKS	Bus cycles when a LOCK signal is asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: - Uncacheable memory. - Locked operation that spans two cache lines. - Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
64H	See Table 18-61.	BUS_DATA_RCV	Bus cycles while processor receives data.	This event counts the number of cycles during which the processor is busy receiving data. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
65H	See Table 18-61 and Table 18-62.	BUS_TRANS_BRD	Burst read bus transactions.	This event counts the number of burst read transactions including: - L1 data cache read misses (and L1 data cache hardware prefetches). - L2 hardware prefetches by the DPL and L2 streamer. - IFU read misses of cacheable lines. It does not include RFO transactions.
66H	See Table 18-61 and Table 18-62.	BUS_TRANS_RFO	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. This event also counts RFO bus transactions due to locked operations.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
67H	See Table 18-61 and Table 18-62.	BUS_TRANS_WB	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-61 and Table 18-62.	BUS_TRANS_IFETCH	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-61 and Table 18-62.	BUS_TRANS_INVAL	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: - A store operation hits a shared line in the L2 cache. - A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-61 and Table 18-62.	BUS_TRANS_PWR	Partial write bus transaction.	This event counts partial write bus transactions.
6BH	See Table 18-61 and Table 18-62.	BUS_TRANS_P	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-61 and Table 18-62.	BUS_TRANS_IO	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-61 and Table 18-62.	BUS_TRANS_DEF	Deferred bus transactions.	This event counts the number of deferred transactions.
6EH	See Table 18-61 and Table 18-62.	BUS_TRANS_BURST	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: - Burst reads. - RFOs. - Explicit writebacks. - Write combine lines.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
6FH	See Table 18-61 and Table 18-62.	BUS_TRANS_MEM	Memory bus transactions.	This event counts all memory bus transactions including: - Burst transactions. - Partial reads and writes. - Invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_INVAL.
70H	See Table 18-61 and Table 18-62.	BUS_TRANS_ANY	All bus transactions.	This event counts all bus transactions. This includes: - Memory transactions. - IO transactions (non memory-mapped). - Deferred transaction completion. - Other less frequent transactions, such as interrupts.
77H	See Table 18-61 and Table 18-64.	EXT_SNOOP	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7AH	See Table 18-62.	BUS_HIT_DRV	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7BH	See Table 18-62.	BUS_HITM_DRV	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7DH	See Table 18-61.	BUSQ_EMPTY	Bus queue is empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7EH	See Table 18-61 and Table 18-62.	SNOOP_STALL_DRV	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. During the snoop stall cycles no new bus transactions requiring a snoop response can be initiated on the bus. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7FH	See Table 18-61.	BUS_IO_WAIT	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. This event counts IO requests from the core.
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	04H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes.
82H	02H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches that miss the ITLB.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
AAH	02H	MACRO_INSTS.CISC_DECODED	CISC macro instructions decoded.	This event counts the number of complex instructions decoded, but not necessarily executed or retired. Only one complex instruction can be decoded at a time.
AAH	03H	MACRO_INSTS.ALL_DECODED	All Instructions decoded.	This event counts the number of instructions decoded.
B0H	00H	SIMD_UOPS_EXEC.S	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. This event does not count MOVQ and MOVD stores from register to memory.
B0H	80H	SIMD_UOPS_EXEC.AR	SIMD micro-ops retired (excluding stores).	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	00H	SIMD_SAT_UOP_EXEC.S	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	80H	SIMD_SAT_UOP_EXEC.AR	SIMD saturated arithmetic micro-ops retired.	This event counts the number of SIMD saturated arithmetic micro-ops retired.
B3H	01H	SIMD_UOP_TYPE_EXEC.MUL.S	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	81H	SIMD_UOP_TYPE_EXEC.MUL.AR	SIMD packed multiply micro-ops retired.	This event counts the number of SIMD packed multiply micro-ops retired.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHIFT.S	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	82H	SIMD_UOP_TYPE_EXEC.SHIFT.AR	SIMD packed shift micro-ops retired.	This event counts the number of SIMD packed shift micro-ops retired.
B3H	04H	SIMD_UOP_TYPE_EXEC.PACK.S	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	84H	SIMD_UOP_TYPE_EXEC.PACK.AR	SIMD pack micro-ops retired.	This event counts the number of SIMD pack micro-ops retired.
B3H	08H	SIMD_UOP_TYPE_EXEC.UNPACK.S	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	88H	SIMD_UOP_TYPE_EXEC.UNPACK.AR	SIMD unpack micro-ops retired.	This event counts the number of SIMD unpack micro-ops retired.
B3H	10H	SIMD_UOP_TYPE_EXEC.LOGICAL.S	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.
B3H	90H	SIMD_UOP_TYPE_EXEC.LOGICAL.AR	SIMD packed logical micro-ops retired.	This event counts the number of SIMD packed logical micro-ops retired.
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.S	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
B3H	A0H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.AR	SIMD packed arithmetic micro-ops retired.	This event counts the number of SIMD packed arithmetic micro-ops retired.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (precise event).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
N/A	00H	INST_RETIRED.ANY	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
C2H	10H	UOPS_RETIRED.ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. **This is an architectural performance event.**
C4H	01H	BR_INST_RETIRED.PRED_NOT_TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRED_NOT_TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0AH	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
				To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C4H	0FH	BR_INST_RETIRED.ANY1	Retired branch instructions.	This event counts the number of branch instructions retired that were mispredicted. This event is a duplicate of BR_INST_RETIRED.MISPRED.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature. To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C6H	01H	CYCLES_INT_MASKED.CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_MASKED.CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.
C7H	10H	SIMD_INST_RETIRED.VECTOR	Retired Streaming SIMD Extensions 2 (SSE2) vector instructions.	This event counts the number of SSE2 vector instructions retired.
C7H	1FH	SIMD_INST_RETIRED.ANY	Retired Streaming SIMD instructions.	This event counts the overall number of SIMD instructions retired. To count each type of SIMD instruction separately, use the following events: SIMD_INST_RETIRED.PACKED_SINGLE SIMD_INST_RETIRED.SCALAR_SINGLE SIMD_INST_RETIRED.PACKED_DOUBLE SIMD_INST_RETIRED.SCALAR_DOUBLE SIMD_INST_RETIRED.VECTOR.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor. This event will count twice for dual-pipe micro-ops.
CAH	01H	SIMD_COMP_INST_RETIRED.PACKED_SINGLE	Retired computational Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_INST_RETIRED.SCALAR_SINGLE	Retired computational Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_INST_RETIRED.PACKED_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.

Table 19-29. Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
CAH	08H	SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CBH	01H	MEM_LOAD_RETIRED.L2_HIT	Retired loads that hit the L2 cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache and hit the L2 cache.
CBH	02H	MEM_LOAD_RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache.
CBH	04H	MEM_LOAD_RETIRED.DTLB_MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault.
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed after MMX™ technology code has changed the MMX state in the floating point stack. For example, these assists are required in the following cases. Streaming SIMD Extensions (SSE) instructions: 1. Denormal input when the DAZ (Denormals Are Zeros) flag is off. 2. Underflow result when the FTZ (Flush To Zero) flag is off.
CEH	00H	SIMD_INSTR_RETIRED	SIMD Instructions retired.	This event counts the number of SIMD instructions that retired.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
E0H	01H	BR_INST_DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	01H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event and the BTB is flushed. This occurs mainly after task switches.
E6H	01H	BACLEARS.ANY	BACLEARS asserted.	This event counts the number of times the front end is redirected for a branch prediction, mainly when an early branch prediction is corrected by other branch handling mechanisms in the front end. This can occur if the code has many branches such that they cannot be consumed by the branch predictor. Each Baclear asserted costs approximately 7 cycles. The effect on total execution time depends on the surrounding code.

19.17 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Table 19-30 lists model-specific performance events for Intel® Core™ Duo processors. If a model-specific event requires qualification in core specificity, it is indicated in the comment column. Table 19-30 also applies to Intel® Core™ Solo processors; bits in the unit mask corresponding to core-specificity are reserved and should be 00B.

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
03H	LD_Blocks	00H	Load operations delayed due to store buffer blocks. The preceding store may be blocked due to unknown address, unknown data, or conflict due to partial overlap between the load and store.	
04H	SD_Drains	00H	Cycles while draining store buffers.	
05H	Misalign_Mem_Ref	00H	Misaligned data memory references (MOB splits of loads and stores).	
06H	Seg_Reg_Loads	00H	Segment register loads.	
07H	SSE_PrefNta_Ret	00H	SSE software prefetch instruction PREFETCHNTA retired.	
07H	SSE_PrefT1_Ret	01H	SSE software prefetch instruction PREFETCHT1 retired.	
07H	SSE_PrefT2_Ret	02H	SSE software prefetch instruction PREFETCHT2 retired.	
07H	SSE_NTStores_Ret	03H	SSE streaming store instruction retired.	
10H	FP_Comps_Op_Exe	00H	FP computational Instruction executed. FADD, FSUB, FCOM, FMULs, MUL, IMUL, FDIVs, DIV, IDIV, FPREMs, FSQRT are included; but exclude FADD or FMUL used in the middle of a transcendental instruction.	
11H	FP_Assist	00H	FP exceptions experienced microcode assists.	IA32_PMC1 only.
12H	Mul	00H	Multiply operations (a speculative count, including FP and integer multiplies).	IA32_PMC1 only.
13H	Div	00H	Divide operations (a speculative count, including FP and integer divisions).	IA32_PMC1 only.
14H	Cycles_Div_Busy	00H	Cycles the divider is busy.	IA32_PMC0 only.
21H	L2_ADS	00H	L2 Address strobes.	Requires core-specificity.
22H	Dbus_Busy	00H	Core cycle during which data bus was busy (increments by 4).	Requires core-specificity.
23H	Dbus_Busy_Rd	00H	Cycles data bus is busy transferring data to a core (increments by 4).	Requires core-specificity.
24H	L2_Lines_In	00H	L2 cache lines allocated.	Requires core-specificity and HW prefetch qualification.
25H	L2_M_Lines_In	00H	L2 Modified-state cache lines allocated.	Requires core-specificity.
26H	L2_Lines_Out	00H	L2 cache lines evicted.	Requires core-specificity and HW prefetch qualification.
27H	L2_M_Lines_Out	00H	L2 Modified-state cache lines evicted.	

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
28H	L2_IFetch	Requires MESI qualification	L2 instruction fetches from instruction fetch unit (includes speculative fetches).	Requires core-specificity.
29H	L2_LD	Requires MESI qualification	L2 cache reads.	Requires core-specificity.
2AH	L2_ST	Requires MESI qualification	L2 cache writes (includes speculation).	Requires core-specificity.
2EH	L2_Rqsts	Requires MESI qualification	L2 cache reference requests.	Requires core-specificity, HW prefetch qualification.
30H	L2_Reject_Cycles	Requires MESI qualification	Cycles L2 is busy and rejecting new requests.	
32H	L2_No_Request_ Cycles	Requires MESI qualification	Cycles there is no request to access L2.	
3AH	EST_Trans_All	00H	Any Intel Enhanced SpeedStep(R) Technology transitions.	
3AH	EST_Trans_All	10H	Intel Enhanced SpeedStep Technology frequency transitions.	
3BH	Thermal_Trip	C0H	Duration in a thermal trip based on the current core clock.	Use edge trigger to count occurrence.
3CH	NonHlt_Ref_Cycles	01H	Non-halted bus cycles.	
3CH	Serial_Execution_ Cycles	02H	Non-halted bus cycles of this core executing code while the other core is halted.	
40H	DCache_Cache_LD	Requires MESI qualification	L1 cacheable data read operations.	
41H	DCache_Cache_ST	Requires MESI qualification	L1 cacheable data write operations.	
42H	DCache_Cache_ Lock	Requires MESI qualification	L1 cacheable lock read operations to invalid state.	
43H	Data_Mem_Ref	01H	L1 data read and writes of cacheable and non-cacheable types.	
44H	Data_Mem_Cache_ Ref	02H	L1 data cacheable read and write operations.	
45H	DCache_Repl	0FH	L1 data cache line replacements.	
46H	DCache_M_Repl	00H	L1 data M-state cache line allocated.	
47H	DCache_M_Evict	00H	L1 data M-state cache line evicted.	
48H	DCache_Pend_Miss	00H	Weighted cycles of L1 miss outstanding.	Use Cmask =1 to count duration.
49H	Dtlb_Miss	00H	Data references that missed TLB.	
4BH	SSE_PrefNta_Miss	00H	PREFETCHNTA missed all caches.	
4BH	SSE_PrefT1_Miss	01H	PREFETCHT1 missed all caches.	
4BH	SSE_PrefT2_Miss	02H	PREFETCHT2 missed all caches.	
4BH	SSE_NTStores_ Miss	03H	SSE streaming store instruction missed all caches.	
4FH	L1_Pref_Req	00H	L1 prefetch requests due to DCU cache misses.	May overcount if request re-submitted.

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
60H	Bus_Req_ Outstanding	00; Requires core-specificity, and agent specificity	Weighted cycles of cacheable bus data read requests. This event counts full-line read request from DCU or HW prefetcher, but not RFO, write, instruction fetches, or others.	Use Cmask =1 to count duration. Use Umask bit 12 to include HWP or exclude HWP separately.
61H	Bus_BNR_Clocks	00H	External bus cycles while BNR asserted.	
62H	Bus_DRDY_Clocks	00H	External bus cycles while DRDY asserted.	Requires agent specificity.
63H	Bus_Locks_Clocks	00H	External bus cycles while bus lock signal asserted.	Requires core specificity.
64H	Bus_Data_Rcv	40H	Number of data chunks received by this processor.	
65H	Bus_Trans_Brd	See comment.	Burst read bus transactions (data or code).	Requires core specificity.
66H	Bus_Trans_RFO	See comment.	Completed read for ownership (RFO) transactions.	Requires agent specificity. Requires core specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
68H	Bus_Trans_Ifetch	See comment.	Completed instruction fetch transactions.	
69H	Bus_Trans_Inval	See comment.	Completed invalidate transactions.	
6AH	Bus_Trans_Pwr	See comment.	Completed partial write transactions.	
6BH	Bus_Trans_P	See comment.	Completed partial transactions (include partial read + partial write + line write).	
6CH	Bus_Trans_IO	See comment.	Completed I/O transactions (read and write).	
6DH	Bus_Trans_Def	20H	Completed defer transactions.	Requires core specificity. Retried transaction may be counted more than once.
67H	Bus_Trans_WB	C0H	Completed writeback transactions from DCU (does not include L2 writebacks).	Requires agent specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
6EH	Bus_Trans_Burst	C0H	Completed burst transactions (full line transactions include reads, write, RFO, and writebacks).	
6FH	Bus_Trans_Mem	C0H	Completed memory transactions. This includes Bus_Trans_Burst + Bus_Trans_P+Bus_Trans_Inval.	
70H	Bus_Trans_Any	C0H	Any completed bus transactions.	
77H	Bus_Snoops	00H	Counts any snoop on the bus.	Requires MESI qualification. Requires agent specificity.
78H	DCU_Snoop_To_ Share	01H	DCU snoops to share-state L1 cache line due to L1 misses.	Requires core specificity.
7DH	Bus_Not_In_Use	00H	Number of cycles there is no transaction from the core.	Requires core specificity.
7EH	Bus_Snoop_Stall	00H	Number of bus cycles while bus snoop is stalled.	
80H	ICache_Reads	00H	Number of instruction fetches from ICache, streaming buffers (both cacheable and uncacheable fetches).	

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
81H	ICache_Misses	00H	Number of instruction fetch misses from ICache, streaming buffers.	
85H	ITLB_Misses	00H	Number of iITLB misses.	
86H	IFU_Mem_Stall	00H	Cycles IFU is stalled while waiting for data from memory.	
87H	ILD_Stall	00H	Number of instruction length decoder stalls (Counts number of LCP stalls).	
88H	Br_Inst_Exec	00H	Branch instruction executed (includes speculation).	
89H	Br_Missp_Exec	00H	Branch instructions executed and mispredicted at execution (includes branches that do not have prediction or mispredicted).	
8AH	Br_BAC_Missp_ Exec	00H	Branch instructions executed that were mispredicted at front end.	
8BH	Br_Cnd_Exec	00H	Conditional branch instructions executed.	
8CH	Br_Cnd_Missp_ Exec	00H	Conditional branch instructions executed that were mispredicted.	
8DH	Br_Ind_Exec	00H	Indirect branch instructions executed.	
8EH	Br_Ind_Missp_Exec	00H	Indirect branch instructions executed that were mispredicted.	
8FH	Br_Ret_Exec	00H	Return branch instructions executed.	
90H	Br_Ret_Missp_Exec	00H	Return branch instructions executed that were mispredicted.	
91H	Br_Ret_BAC_Missp_ Exec	00H	Return branch instructions executed that were mispredicted at the front end.	
92H	Br_Call_Exec	00H	Return call instructions executed.	
93H	Br_Call_Missp_Exec	00H	Return call instructions executed that were mispredicted.	
94H	Br_Ind_Call_Exec	00H	Indirect call branch instructions executed.	
A2H	Resource_Stall	00H	Cycles while there is a resource related stall (renaming, buffer entries) as seen by allocator.	
B0H	MMX_Instr_Exec	00H	Number of MMX instructions executed (does not include MOVQ and MOVD stores).	
B1H	SIMD_Int_Sat_Exec	00H	Number of SIMD Integer saturating instructions executed.	
B3H	SIMD_Int_Pmul_ Exec	01H	Number of SIMD Integer packed multiply instructions executed.	
B3H	SIMD_Int_Psft_Exec	02H	Number of SIMD Integer packed shift instructions executed.	
B3H	SIMD_Int_Pck_Exec	04H	Number of SIMD Integer pack operations instruction executed.	
B3H	SIMD_Int_Upck_ Exec	08H	Number of SIMD Integer unpack instructions executed.	
B3H	SIMD_Int_Plog_ Exec	10H	Number of SIMD Integer packed logical instructions executed.	
B3H	SIMD_Int_Pari_Exec	20H	Number of SIMD Integer packed arithmetic instructions executed.	

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
C0H	Instr_Ret	00H	Number of instruction retired (Macro fused instruction count as 2).	
C1H	FP_Comp_Instr_Ret	00H	Number of FP compute instructions retired (X87 instruction or instruction that contains X87 operations).	Use IA32_PMC0 only.
C2H	Uops_Ret	00H	Number of micro-ops retired (include fused uops).	
C3H	SMC_Detected	00H	Number of times self-modifying code condition detected.	
C4H	Br_Instr_Ret	00H	Number of branch instructions retired.	
C5H	Br_MisPred_Ret	00H	Number of mispredicted branch instructions retired.	
C6H	Cycles_Int_Masked	00H	Cycles while interrupt is disabled.	
C7H	Cycles_Int_Pedning_ Masked	00H	Cycles while interrupt is disabled and interrupts are pending.	
C8H	HW_Int_Rx	00H	Number of hardware interrupts received.	
C9H	Br_Taken_Ret	00H	Number of taken branch instruction retired.	
CAH	Br_MisPred_Taken_ Ret	00H	Number of taken and mispredicted branch instructions retired.	
CCH	MMX_FP_Trans	00H	Number of transitions from MMX to X87.	
CCH	FP_MMX_Trans	01H	Number of transitions from X87 to MMX.	
CDH	MMX_Assist	00H	Number of EMMS executed.	
CEH	MMX_Instr_Ret	00H	Number of MMX instruction retired.	
D0H	Instr_Decoded	00H	Number of instruction decoded.	
D7H	ESP_Uops	00H	Number of ESP folding instruction decoded.	
D8H	SIMD_FP_SP_Ret	00H	Number of SSE/SSE2 single precision instructions retired (packed and scalar).	
D8H	SIMD_FP_SP_S_ Ret	01H	Number of SSE/SSE2 scalar single precision instructions retired.	
D8H	SIMD_FP_DP_P_ Ret	02H	Number of SSE/SSE2 packed double precision instructions retired.	
D8H	SIMD_FP_DP_S_ Ret	03H	Number of SSE/SSE2 scalar double precision instructions retired.	
D8H	SIMD_Int_128_Ret	04H	Number of SSE2 128 bit integer instructions retired.	
D9H	SIMD_FP_SP_P_ Comp_Ret	00H	Number of SSE/SSE2 packed single precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_SP_S_ Comp_Ret	01H	Number of SSE/SSE2 scalar single precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_DP_P_ Comp_Ret	02H	Number of SSE/SSE2 packed double precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_DP_S_ Comp_Ret	03H	Number of SSE/SSE2 scalar double precision compute instructions retired (does not include AND, OR, XOR).	

Table 19-30. Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
DAH	Fused_Uops_Ret	00H	All fused uops retired.	
DAH	Fused_Ld_Uops_ Ret	01H	Fused load uops retired.	
DAH	Fused_St_Uops_Ret	02H	Fused store uops retired.	
DBH	Unfusion	00H	Number of unfusion events in the ROB (due to exception).	
E0H	Br_Instr_Decoded	00H	Branch instructions decoded.	
E2H	BTB_Misses	00H	Number of branches the BTB did not produce a prediction.	
E4H	Br_Bogus	00H	Number of bogus branches.	
E6H	BAClears	00H	Number of BAClears asserted.	
F0H	Pref_Rqsts_Up	00H	Number of hardware prefetch requests issued in forward streams.	
F8H	Pref_Rqsts_Dn	00H	Number of hardware prefetch requests issued in backward streams.	

19.18 PENTIUM® 4 AND INTEL® XEON® PROCESSOR PERFORMANCE MONITORING EVENTS

Tables 19-31, 19-32 and 19-33 list performance monitoring events that can be counted or sampled on processors based on Intel NetBurst® microarchitecture. Table 19-31 lists the non-retirement events, and Table 19-32 lists the at-retirement events. Tables 19-34, 19-35, and 19-36 describes three sets of parameters that are available for three of the at-retirement counting events defined in Table 19-32. Table 19-37 shows which of the non-retirement and at retirement events are logical processor specific (TS) (see Section 18.6.4.4, "Performance Monitoring Events") and which are non-logical processor specific (TI).

Some of the Pentium 4 and Intel Xeon processor performance monitoring events may be available only to specific models. The performance monitoring events listed in Tables 19-31 and 19-32 apply to processors with CPUID signature that matches family encoding 15, model encoding 0, 1, 2 3, 4, or 6. Table applies to processors with a CPUID signature that matches family encoding 15, model encoding 3, 4 or 6.

The functionality of performance monitoring events in Pentium 4 and Intel Xeon processors is also available when IA-32e mode is enabled.

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
TC_deliver_mode			This event counts the duration (in clock cycles) of the operating modes of the trace cache and decode engine in the processor package. The mode is specified by one or more of the event mask bits.
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	01H	ESCR[31:25]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit	ESCR[24:9]
		0: DD	Both logical processors are in deliver mode.
		1: DB	Logical processor 0 is in deliver mode and logical processor 1 is in build mode.
		2: DI	Logical processor 0 is in deliver mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow.
		3: BD	Logical processor 0 is in build mode and logical processor 1 is in deliver mode.
		4: BB	Both logical processors are in build mode.
		5: BI	Logical processor 0 is in build mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow.
		6: ID	Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in deliver mode.
		7: IB	Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in build mode.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If only one logical processor is available from a physical processor package, the event mask should be interpreted as logical processor 1 is halted. Event mask bit 2 was previously known as "DELIVER", bit 5 was previously known as "BUILD".
BPU_fetch_ request			This event counts instruction fetch requests of specified request type by the Branch Prediction unit. Specify one or more mask bits to qualify the request type(s).
	ESCR restrictions	MSR_BPU_ESCR0 MSR_BPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: TCMISS	ESCR[24:9] Trace cache lookup miss
	CCCR Select	00H	CCCR[15:13]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
ITLB_reference			This event counts translations using the Instruction Translation Look-aside Buffer (ITLB).
	ESCR restrictions	MSR_ITLB_ESCR0 MSR_ITLB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	18H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: HIT 1: MISS 2: HIT_UC	ESCR[24:9] ITLB hit ITLB miss Uncacheable ITLB hit
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		All page references regardless of the page size are looked up as actual 4-KByte pages. Use the page_walk_type event with the ITMISS mask for a more conservative count.
memory_cancel			This event counts the canceling of various type of request in the Data cache Address Control unit (DAC). Specify one or more mask bits to select the type of requests that are canceled.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 2: ST_RB_FULL 3: 64K_CONF	ESCR[24:9] Replayed because no store request buffer is available. Conflicts due to 64-KByte aliasing.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		All_CACHE_MISS includes uncacheable memory in count.
memory_ complete			This event counts the completion of a load split, store split, uncacheable (UC) split, or UC load. Specify one or more mask bits to select the operations to be counted.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: LSC 1: SSC	ESCR[24:9] Load split completed, excluding UC/WC loads. Any split stores completed.
	CCCR Select	02H	CCCR[15:13]
load_port_replay			This event counts replayed events at the load port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_LD	ESCR[24:9] Split load.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
store_port_replay			This event counts replayed events at the store port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_ST	ESCR[24:9] Split store
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
MOB_load_replay			This event triggers if the memory order buffer (MOB) caused a load operation to be replayed. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_MOB_ESCR0 MSR_MOB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 1: NO_STA 3: NO_STD	ESCR[24:9] Replayed because of unknown store address. Replayed because of unknown store data.
		4: PARTIAL_DATA 5: UNALGN_ADDR	Replayed because of partially overlapped data access between the load and store operations. Replayed because the lower 4 bits of the linear address do not match between the load and store operations.
	CCCR Select	02H	CCCR[15:13]
page_walk_type			This event counts various types of page walks that the page miss handler (PMH) performs.
	ESCR restrictions	MSR_PMH_ESCR0 MSR_PMH_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	Bit 0: DTMISS 1: ITMISS	ESCR[24:9] Page walk for a data TLB miss (either load or store). Page walk for an instruction TLB miss.
	CCCR Select	04H	CCCR[15:13]
BSQ_cache_reference			This event counts cache references (2nd level cache or 3rd level cache) as seen by the bus unit. Specify one or more mask bit to select an access according to the access type (read type includes both load and RFO, write type includes writebacks and evictions) and the access result (hit, misses).
	ESCR restrictions	MSR_BSU_ESCR0 MSR_BSU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	0CH	ESCR[31:25]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		Bit	ESCR[24:9]
		0: RD_2ndL_HITS	Read 2nd level cache hit Shared (includes load and RFO).
		1: RD_2ndL_HITE	Read 2nd level cache hit Exclusive (includes load and RFO).
		2: RD_2ndL_HITM	Read 2nd level cache hit Modified (includes load and RFO).
		3: RD_3rdL_HITS	Read 3rd level cache hit Shared (includes load and RFO).
		4: RD_3rdL_HITE	Read 3rd level cache hit Exclusive (includes load and RFO).
		5: RD_3rdL_HITM	Read 3rd level cache hit Modified (includes load and RFO).
	ESCR Event Mask	8: RD_2ndL_MISS	Read 2nd level cache miss (includes load and RFO).
		9: RD_3rdL_MISS	Read 3rd level cache miss (includes load and RFO).
		10: WR_2ndL_MISS	A Writeback lookup from DAC misses the 2nd level cache (unlikely to happen).
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: The implementation of this event in current Pentium 4 and Xeon processors treats either a load operation or a request for ownership (RFO) request as a "read" type operation. 2: Currently this event causes both over and undercounting by as much as a factor of two due to an erratum. 3: It is possible for a transaction that is started as a prefetch to change the transaction's internal status, making it no longer a prefetch. or change the access result status (hit, miss) as seen by this event.
IOQ_allocation			This event counts the various types of transactions on the bus. A count is generated each time a transaction is allocated into the IOQ that matches the specified mask bits. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. Requests are counted once per retry. The event mask bits constitute 4 bit fields. A transaction type is specified by interpreting the values of each bit field. Specify one or more event mask bits in a bit field to select the value of the bit field. Each field (bits 0-4 are one field) are independent of and can be ORed with the others. The request type field is further combined with bit 5 and 6 to form a binary expression. Bits 7 and 8 form a bit field to specify the memory type of the target address. Bits 13 and 14 form a bit field to specify the source agent of the request. Bit 15 affects read operation only. The event is triggered by evaluating the logical expression: (((Request type) OR Bit 5 OR Bit 6) OR (Memory type)) AND (Source agent).

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR restrictions	MSR_FSB_ESCR0, MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1; ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bits	
		0-4 (single field)	Bus request type (use 00001 for invalid or default).
		5: ALL_READ	Count read entries.
		6: ALL_WRITE	Count write entries.
		7: MEM_UC	Count UC memory access entries.
		8: MEM_WC	Count WC memory access entries.
		9: MEM_WT	Count write-through (WT) memory access entries.
		10: MEM_WP	Count write-protected (WP) memory access entries.
		11: MEM_WB	Count WB memory access entries.
		13: OWN	Count all store requests driven by processor, as opposed to other processor or DMA.
		14: OTHER	Count all requests driven by other processors or DMA.
		15: PREFETCH	Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: If PREFETCH bit is cleared, sectors fetched using prefetch are excluded in the counts. If PREFETCH bit is set, all sectors or chunks read are counted. 2: Specify the edge trigger in CCCR to avoid double counting.
			3: The mapping of interpreted bit field values to transaction types may differ with different processor model implementations of the Pentium 4 processor family. Applications that program performance monitoring events should use CPUID to determine processor models when using this event. The logic equations that trigger the event are model-specific (see 4a and 4b below). 4a: For Pentium 4 and Xeon Processors starting with CPUID Model field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)). 4b: For Pentium 4 and Xeon Processors with CPUID Model field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Note that event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5: This event is known to ignore CPL in early implementations of Pentium 4 and Xeon Processors. Both user requests and OS requests are included in the count. This behavior is fixed starting with Pentium 4 and Xeon Processors with CPUID signature F27H (Family 15, Model 2, Stepping 7).

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			6: For write-through (WT) and write-protected (WP) memory types, this event counts reads as the number of 64-byte sectors. Writes are counted by individual chunks. 7: For uncacheable (UC) memory types, this event counts the number of 8-byte chunks allocated. 8: For Pentium 4 and Xeon Processors with CPUID Signature less than F27H, only MSR_FSB_ESCR0 is available.
IOQ_active_ entries			This event counts the number of entries (clipped at 15) in the IOQ that are active. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. The event must be programmed in conjunction with IOQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	01AH	ESCR[30:25]
	ESCR Event Mask	Bits 0-4 (single field) 5: ALL_READ 6: ALL_WRITE 7: MEM_UC 8: MEM_WC 9: MEM_WT 10: MEM_WP 11: MEM_WB 13: OWN 14: OTHER 15: PREFETCH	ESCR[24:9] Bus request type (use 00001 for invalid or default). Count read entries. Count write entries. Count UC memory access entries. Count WC memory access entries. Count write-through (WT) memory access entries. Count write-protected (WP) memory access entries. Count WB memory access entries. Count all store requests driven by processor, as opposed to other processor or DMA. Count all requests driven by other processors or DMA. Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the ioq_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles. 4: The mapping of interpreted bit field values to transaction types may differ across different processor model implementations of the Pentium 4 processor family. Applications that programs performance monitoring events should use the CPUID instruction to detect processor models when using this event. The logical expression that triggers this event as describe below: 5a:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)).

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5b:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5c: This event is known to ignore CPL in the current implementations of Pentium 4 and Xeon Processors Both user requests and OS requests are included in the count.
			6: An allocated entry can be a full line (64 bytes) or in individual chunks of 8 bytes.
FSB_data_ activity			This event increments once for each DRDY or DBSY event that occurs on the front side bus. The event allows selection of a specific DRDY or DBSY event.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	17H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit 0: DRDY_DRV	Count when this processor drives data onto the bus - includes writes and implicit writebacks. Asserted two processor clock cycles for partial writes and 4 processor clocks (usually in consecutive bus clocks) for full line writes.
		1: DRDY_OWN	Count when this processor reads data from the bus - includes loads and some PIC transactions. Asserted two processor clock cycles for partial reads and 4 processor clocks (usually in consecutive bus clocks) for full line reads. Count DRDY events that we drive. Count DRDY events sampled that we own.
		2: DRDY_OTHER	Count when data is on the bus but not being sampled by the processor. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
		3: DBSY_DRV	Count when this processor reserves the bus for use in the next bus cycle in order to drive data. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (in consecutive bus clocks) if we stall the bus waiting for a cache lock to complete.
		4: DBSY_OWN	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will sample. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (all one bus clock apart) if we stall the bus for some reason.

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		5:DBSY_OTHER	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will NOT sample. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		Specify edge trigger in the CCCR MSR to avoid double counting. DRDY_OWN and DRDY_OTHER are mutually exclusive; similarly for DBSY_OWN and DBSY_OTHER.
BSQ_allocation			This event counts allocations in the Bus Sequence Unit (BSQ) according to the specified mask bit encoding. The event mask bits consist of four sub-groups: ▪ Request type. ▪ Request length. ▪ Memory type. ▪ Sub-group consisting mostly of independent bits (bits 5, 6, 7, 8, 9, and 10). Specify an encoding for each sub-group.
	ESCR restrictions	MSR_BSU_ESCR0	
	Counter numbers per ESCR	ESCR0: 0, 1	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit	ESCR[24:9]
		0: REQ_TYPE0 1: REQ_TYPE1	Request type encoding (bit 0 and 1) are: 0 – Read (excludes read invalidate). 1 – Read invalidate. 2 – Write (other than writebacks). 3 – Writeback (evicted from cache). (public)
		2: REQ_LEN0 3: REQ_LEN1	Request length encoding (bit 2, 3) are: 0 – 0 chunks 1 – 1 chunks 3 – 8 chunks
		5: REQ_IO_TYPE	Request type is input or output.
		6: REQ_LOCK_ TYPE	Request type is bus lock.
		7: REQ_CACHE_ TYPE	Request type is cacheable.
		8: REQ_SPLIT_ TYPE	Request type is a bus 8-byte chunk split across 8-byte boundary.
		9: REQ_DEM_TYPE	Request type is a demand if set. Request type is HW.SW prefetch if 0.
		10: REQ_ORD_ TYPE	Request is an ordered type.

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		11: MEM_TYPE0 12: MEM_TYPE1 13: MEM_TYPE2	Memory type encodings (bit 11-13) are: 0 – UC 1 – WC 4 – WT 5 – WP 6 – WB
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specify edge trigger in CCCR to avoid double counting. 2: A writebacks to 3rd level cache from 2nd level cache counts as a separate entry, this is in additional to the entry allocated for a request to the bus. 3: A read request to WB memory type results in a request to the 64-byte sector, containing the target address, followed by a prefetch request to an adjacent sector. 4: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 0 and 1, an allocated BSQ entry includes both the demand sector and prefetched 2nd sector. 5: An allocated BSQ entry for a data chunk is any request less than 64 bytes. 6a: This event may undercount for requests of split type transactions if the data address straddled across modulo-64 byte boundary. 6b: This event may undercount for requests of read request of 16-byte operands from WC or UC address. 6c: This event may undercount WC partial requests originated from store operands that are dwords.
bsq_active_ entries			This event represents the number of BSQ entries (clipped at 15) currently active (valid) which meet the subevent mask criteria during allocation in the BSQ. Active request entries are allocated on the BSQ until de-allocated. De-allocation of an entry does not necessarily imply the request is filled. This event must be programmed in conjunction with BSQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	ESCR Event Mask		ESCR[24:9]
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the BSQ_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles.
			4: This event can be used to estimate the latency of a transaction from allocation to de-allocation in the BSQ. The latency observed by BSQ_allocation includes the latency of FSB, plus additional overhead.

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5: Additional overhead may include the time it takes to issue two requests (the sector by demand and the adjacent sector via prefetch). Since adjacent sector prefetches have lower priority that demand fetches, on a heavily used system there is a high probability that the adjacent sector prefetch will have to wait until the next bus arbitration. 6: For Pentium 4 and Xeon processors with CPUID model encoding value less than 3, this event is updated every clock. 7: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 3 or 4, this event is updated every other clock.
SSE_input_assist			This event counts the number of times an assist is requested to handle problems with input operands for SSE/SSE2/SSE3 operations; most notably denormal source operands when the DAZ bit is not set. Set bit 15 of the event mask to use this event.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	34H	ESCR[31:25]
	ESCR Event Mask	15: ALL	ESCR[24:9] Count assists for SSE/SSE2/SSE3 μops.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: Not all requests for assists are actually taken. This event is known to overcount in that it counts requests for assists from instructions on the non-retired path that do not incur a performance penalty. An assist is actually taken only for non-bogus μops. Any appreciable counts for this event are an indication that the DAZ or FTZ bit should be set and/or the source code should be changed to eliminate the condition. 2: Two common situations for an SSE/SSE2/SSE3 operation needing an assist are: (1) when a denormal constant is used as an input and the Denormals-Are-Zero (DAZ) mode is not set, (2) when the input operand uses the underflowed result of a previous SSE/SSE2/SSE3 operation and neither the DAZ nor Flush-To-Zero (FTZ) modes are set. 3: Enabling the DAZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the first situation. Enabling the FTZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the second situation.
packed_SP_uop			This event increments for each packed single-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on packed single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: If an instruction contains more than one packed SP μops, each packed SP μop that is specified by the event mask will be counted. 2: This metric counts instances of packed memory μops in a repeat move string.
packed_DP_uop			This event increments for each packed double-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on packed double-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one packed DP μops, each packed DP μop that is specified by the event mask will be counted.
scalar_SP_uop			This event increments for each scalar single-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on scalar single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one scalar SP μops, each scalar SP μop that is specified by the event mask will be counted.
scalar_DP_uop			This event increments for each scalar double-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0EH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on scalar double-precision operands.
	CCCR Select	01H	CCCR[15:13]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		If an instruction contains more than one scalar DP μops, each scalar DP μop that is specified by the event mask is counted.
64bit_MMX_uop			This event increments for each MMX instruction, which operate on 64-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 64- bit SIMD integer operands in memory or MMX registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 64-bit MMX μops, each 64-bit MMX μop that is specified by the event mask will be counted.
128bit_MMX_uop			This event increments for each integer SIMD SSE2 instruction, which operate on 128-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	1AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 128-bit SIMD integer operands in memory or XMM registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 128-bit MMX μops, each 128-bit MMX μop that is specified by the event mask will be counted.
x87_FP_uop			This event increments for each x87 floating-point μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all x87 FP μops.
	CCCR Select	01H	CCCR[15:13]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		1: If an instruction contains more than one x87 FP μops, each x87 FP μop that is specified by the event mask will be counted. 2: This event does not count x87 FP μop for load, store, move between registers.
TC_misc			This event counts miscellaneous events detected by the TC. The counter will count twice for each occurrence.
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	06H	ESCR[31:25]
	CCCR Select	01H	CCCR[15:13]
	ESCR Event Mask	Bit 4: FLUSH	ESCR[24:9] Number of flushes
global_power_events			This event accumulates the time during which a processor is not stopped.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	013H	ESCR[31:25]
	ESCR Event Mask	Bit 0: Running	ESCR[24:9] The processor is active (includes the handling of HLT STPCLK and throttling.
	CCCR Select	06H	CCCR[15:13]
tc_ms_xfer			This event counts the number of times that uop delivery changed from TC to MS ROM.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit 0: CISC	ESCR[24:9] A TC to MS transfer occurred.
	CCCR Select	0H	CCCR[15:13]
uop_queue_writes			This event counts the number of valid uops written to the uop queue. Specify one or more mask bits to select the source type of writes.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FROM_TC_ BUILD 1: FROM_TC_ DELIVER 2: FROM_ROM	ESCR[24:9] The uops being written are from TC build mode. The uops being written are from TC deliver mode. The uops being written are from microcode ROM.
	CCCR Select	0H	CCCR[15:13]
retired_mispred _branch_type			This event counts retiring mispredicted branches by type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL	ESCR[24:9] Conditional jumps. Indirect call branches.
		3: RETURN 4: INDIRECT	Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		This event may overcount conditional branches if: ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
retired_branch _type			This event counts retiring branches by type. Specify one or more mask bits to qualify the branch by its type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	04H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL 3: RETURN 4: INDIRECT	ESCR[24:9] Conditional jumps. Direct or indirect calls. Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		This event may overcount conditional branches if : ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
resource_stall			This event monitors the occurrence or latency of stalls in the Allocator.
	ESCR restrictions	MSR_ALF_ESCR0 MSR_ALF_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[30:25]
	Event Masks	Bit 5: SBFULL	ESCR[24:9] A Stall due to lack of store buffers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
WC_Buffer			This event counts Write Combining Buffer operations that are selected by the event mask.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[30:25]
	Event Masks	Bit 0: WCB_EVICTS	ESCR[24:9] WC Buffer evictions of all causes.
		1: WCB_FULL_ EVICT	WC Buffer eviction: no WC buffer is available.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		This event is useful for detecting the subset of 64K aliasing cases that are more costly (i.e. 64K aliasing cases involving stores) as long as there are no significant contributions due to write combining buffer full or hit-modified conditions.
b2b_cycles			This event can be configured to count the number back-to-back bus cycles using sub-event mask bits 1 through 6.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	016H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]

Table 19-31. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
bnr			This event can be configured to count bus not ready conditions using sub-event mask bits 0 through 2.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	08H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
snoop			This event can be configured to count snoop hit modified bus traffic using sub-event mask bits 2, 6 and 7.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
Response			This event can be configured to count different types of responses using sub-event mask bits 1,2, 8, and 9.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	04H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.

Table 19-32. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
front_end_event			This event counts the retirement of tagged μops, which are specified through the front-end tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	08H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	Selected ESCRs and/or MSR_TC_PRECISE_EVENT	See list of metrics supported by Front_end tagging in Table A-3
execution_event			This event counts the retirement of tagged μops, which are specified through the execution tagging mechanism. The event mask allows from one to four types of μops to be specified as either bogus or non-bogus μops to be tagged.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS0 1: NBOGUS1 2: NBOGUS2 3: NBOGUS3 4: BOGUS0 5: BOGUS1 6: BOGUS2 7: BOGUS3	ESCR[24:9] The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Each of the 4 slots to specify the bogus/non-bogus μops must be coordinated with the 4 TagValue bits in the ESCR (for example, NBOGUS0 must accompany a '1' in the lowest bit of the TagValue field in ESCR, NBOGUS1 must accompany a '1' in the next but lowest bit of the TagValue field).
	Can Support PEBS	Yes	

Table 19-32. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Require Additional MSRs for tagging	An ESCR for an upstream event	See list of metrics supported by execution tagging in Table A-4.
replay_event			This event counts the retirement of tagged μops, which are specified through the replay tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Supports counting tagged μops with additional MSRs.
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	IA32_PEBS_ ENABLE MSR_PEBS_ MATRIX_VERT Selected ESCR	See list of metrics supported by replay tagging in Table A-5.
instr_retired			This event counts instructions that are retired during a clock cycle. Mask bits specify bogus or non-bogus (and whether they are tagged using the front-end tagging mechanism).
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUSNTAG 1: NBOGUSTAG 2: BOGUSNTAG 3: BOGUSTAG	ESCR[24:9] Non-bogus instructions that are not tagged. Non-bogus instructions that are tagged. Bogus instructions that are not tagged. Bogus instructions that are tagged.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		1: The event count may vary depending on the microarchitectural states of the processor when the event detection is enabled. 2: The event may count more than once for some instructions with complex uop flows and were interrupted before retirement.

Table 19-32. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Can Support PEBS	No	
uops_retired			This event counts μops that are retired during a clock cycle. Mask bits specify bogus or non-bogus.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		P6: EMON_UOPS_RETIRED
	Can Support PEBS	No	
uop_type			This event is used in conjunction with the front-end at-retirement mechanism to tag load and store μops.
	ESCR restrictions	MSR_RAT_ESCR0 MSR_RAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 1: TAGLOADS 2: TAGSTORES	ESCR[24:9] The μop is a load operation. The μop is a store operation.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Setting the TAGLOADS and TAGSTORES mask bits does not cause a counter to increment. They are only used to tag uops.
	Can Support PEBS	No	
branch_retired			This event counts the retirement of a branch. Specify one or more mask bits to select any combination of taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 18-70 for the addresses of the ESCR MSRs
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 18-70.
	ESCR Event Select	06H	ESCR[31:25]

Table 19-32. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch not-taken predicted Branch not-taken mispredicted Branch taken predicted Branch taken mispredicted
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
mispred_branch_ retired			This event represents the retirement of mispredicted branch instructions.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS	ESCR[24:9] The retired instruction is not bogus.
	CCCR Select	04H	CCCR[15:13]
	Can Support PEBS	No	
x87_assist			This event counts the retirement of x87 instructions that required special handling. Specifies one or more event mask bits to select the type of assistance.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FPSU 1: FPSO 2: POAO 3: POAU 4: PREA	ESCR[24:9] Handle FP stack underflow. Handle FP stack overflow. Handle x87 output overflow. Handle x87 output underflow. Handle x87 input assist.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-32. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
machine_clear			This event increments according to the mask bit specified while the entire pipeline of the machine is cleared. Specify one of the mask bit to select the cause.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 0: CLEAR 2: MOCLEAR 6: SMCLEAR	ESCR[24:9] Counts for a portion of the many cycles while the machine is cleared for any cause. Use Edge triggering for this bit only to get a count of occurrence versus a duration. Increments each time the machine is cleared due to memory ordering issues. Increments each time the machine is cleared due to self-modifying code issues.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-33. Intel NetBurst® Microarchitecture Model-Specific Performance Monitoring Events (For Model Encoding 3, 4 or 6)

Event Name	Event Parameters	Parameter Value	Description
instr_completed			This event counts instructions that have completed and retired during a clock cycle. Mask bits specify whether the instruction is bogus or non-bogus and whether they are:
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	07H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] Non-bogus instructions Bogus instructions
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		This metric differs from instr_retired, since it counts instructions completed, rather than the number of times that instructions started.
	Can Support PEBS	No	

Table 19-34. List of Metrics Available for Front_end Tagging (For Front_end Event Only)

Front-end metric[1]	MSR_ TC_PRECISE_EVENT MSR Bit field	Additional MSR	Event mask value for Front_end_event
memory_loads	None	Set TAGLOADS bit in ESCR corresponding to event Uop_Type.	NBOGUS
memory_stores	None	Set TAGSTORES bit in the ESCR corresponding to event Uop_Type.	NBOGUS

NOTES:

1. There may be some undercounting of front end events when there is an overflow or underflow of the floating point stack.

Table 19-35. List of Metrics Available for Execution Tagging (For Execution Event Only)

Execution metric	Upstream ESCR	TagValue in Upstream ESCR	Event mask value for execution_event
packed_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_SP_uop.	1	NBOGUS0
packed_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_DP_uop.	1	NBOGUS0
scalar_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_SP_uop.	1	NBOGUS0
scalar_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_DP_uop.	1	NBOGUS0
128_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 128_bit_MMX_uop.	1	NBOGUS0
64_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 64_bit_MMX_uop.	1	NBOGUS0
X87_FP_retired	Set ALL bit in event mask, TagUop bit in ESCR of x87_FP_uop.	1	NBOGUS0
X87_SIMD_memory_moves_retired	Set ALLP0, ALLP2 bits in event mask, TagUop bit in ESCR of X87_SIMD_ moves_uop.	1	NBOGUS0

Table 19-36. List of Metrics Available for Replay Tagging (For Replay Event Only)

Replay metric[1]	IA32_PEBS_ ENABLE Field to Set	MSR_PEBS_ MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
1stL_cache_load _miss_retired	Bit 0, Bit 24, Bit 25	Bit 0	None	NBOGUS
2ndL_cache_load _miss_retired[2]	Bit 1, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_load_miss _retired	Bit 2, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_store_miss _retired	Bit 2, Bit 24, Bit 25	Bit 1	None	NBOGUS
DTLB_all_miss _retired	Bit 2, Bit 24, Bit 25	Bit 0, Bit 1	None	NBOGUS
Tagged_mispred_ branch	Bit 15, Bit 16, Bit 24, Bit 25	Bit 4	None	NBOGUS

Table 19-36. List of Metrics Available for Replay Tagging (For Replay Event Only) (Contd.)

Replay metric[1]	IA32_PEBS_ ENABLE Field to Set	MSR_PEBS_ MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
MOB_load _replay_retired[3]	Bit 9, Bit 24, Bit 25	Bit 0	Select MOB_load_replay event and set PARTIAL_DATA and UNALGN_ADDR bit.	NBOGUS
split_load_retired	Bit 10, Bit 24, Bit 25	Bit 0	Select load_port_replay event with the MSR_SAAT_ESCR1 MSR and set the SPLIT_LD mask bit.	NBOGUS
split_store_retired	Bit 10, Bit 24, Bit 25	Bit 1	Select store_port_replay event with the MSR_SAAT_ESCR0 MSR and set the SPLIT_ST mask bit.	NBOGUS

NOTES:

1. Certain kinds of μops cannot be tagged. These include I/O operations, UC and locked accesses, returns, and far transfers.
2. 2nd-level misses retired does not count all 2nd-level misses. It only includes those references that are found to be misses by the fast detection logic and not those that are later found to be misses.
3. While there are several causes for a MOB replay, the event counted with this event mask setting is the case where the data from a load that would otherwise be forwarded is not an aligned subset of the data from a preceding store.

Table 19-37. Event Mask Qualification for Logical Processors

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	BPU_fetch_request	Bit 0: TCMISS	TS
Non-Retirement	BSQ_allocation	Bit	
		0: REQ_TYPE0	TS
		1: REQ_TYPE1	TS
		2: REQ_LEN0	TS
		3: REQ_LEN1	TS
		5: REQ_IO_TYPE	TS
		6: REQ_LOCK_TYPE	TS
		7: REQ_CACHE_TYPE	TS
		8: REQ_SPLIT_TYPE	TS
		9: REQ_DEM_TYPE	TS
		10: REQ_ORD_TYPE	TS
		11: MEM_TYPE0	TS
		12: MEM_TYPE1	TS
		13: MEM_TYPE2	TS
Non-Retirement	BSQ_cache_reference	Bit	
		0: RD_2ndL_HITS	TS
		1: RD_2ndL_HITE	TS
		2: RD_2ndL_HITM	TS
		3: RD_3rdL_HITS	TS
		4: RD_3rdL_HITE	TS
		5: RD_3rdL_HITM	TS
		6: WR_2ndL_HIT	TS
		7: WR_3rdL_HIT	TS
		8: RD_2ndL_MISS	TS
		9: RD_3rdL_MISS	TS
		10: WR_2ndL_MISS	TS
		11: WR_3rdL_MISS	TS
Non-Retirement	memory_cancel	Bit	
		2: ST_RB_FULL	TS
		3: 64K_CONF	TS
Non-Retirement	SSE_input_assist	Bit 15: ALL	TI
Non-Retirement	64bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	packed_DP_uop	Bit 15: ALL	TI
Non-Retirement	packed_SP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_DP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_SP_uop	Bit 15: ALL	TI
Non-Retirement	128bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	x87_FP_uop	Bit 15: ALL	TI

Table 19-37. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	x87_SIMD_moves_uop	Bit	
		3: ALLP0	TI
		4: ALLP2	TI
Non-Retirement	FSB_data_activity	Bit	
		0: DRDY_DRV	TI
		1: DRDY_OWN	TI
		2: DRDY_OTHER	TI
		3: DBSY_DRV	TI
		4: DBSY_OWN	TI
		5: DBSY_OTHER	TI
Non-Retirement	IOQ_allocation	Bit	
		0: ReqA0	TS
		1: ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS
		13: OWN	TS
		14: OTHER	TS
		15: PREFETCH	TS
Non-Retirement	IOQ_active_entries	Bit	TS
		0: ReqA0	
		1:ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS

Table 19-37. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		13: OWN	TS
		14: OTHER	TS
		15: PREFETCH	TS
Non-Retirement	global_power_events	Bit 0: RUNNING	TS
Non-Retirement	ITLB_reference	Bit	
		0: HIT	TS
		1: MISS	TS
		2: HIT_UC	TS
Non-Retirement	MOB_load_replay	Bit	
		1: NO_STA	TS
		3: NO_STD	TS
		4: PARTIAL_DATA	TS
		5: UNALGN_ADDR	TS
Non-Retirement	page_walk_type	Bit	
		0: DTMISS	TI
		1: ITMISS	TI
Non-Retirement	uop_type	Bit	
		1: TAGLOADS	TS
		2: TAGSTORES	TS
Non-Retirement	load_port_replay	Bit 1: SPLIT_LD	TS
Non-Retirement	store_port_replay	Bit 1: SPLIT_ST	TS
Non-Retirement	memory_complete	Bit	
		0: LSC	TS
		1: SSC	TS
		2: USC	TS
		3: ULC	TS
Non-Retirement	retired_mispred_branch_ type	Bit	
		0: UNCONDITIONAL	TS
		1: CONDITIONAL	TS
		2: CALL	TS
		3: RETURN	TS
		4: INDIRECT	TS
Non-Retirement	retired_branch_type	Bit	
		0: UNCONDITIONAL	TS
		1: CONDITIONAL	TS
		2: CALL	TS
		3: RETURN	TS
		4: INDIRECT	TS

Table 19-37. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	tc_ms_xfer	Bit 0: CISC	TS
Non-Retirement	tc_misc	Bit 4: FLUSH	TS
Non-Retirement	TC_deliver_mode	Bit 0: DD 1: DB 2: DI 3: BD 4: BB 5: BI 6: ID 7: IB	TI TI TI TI TI TI TI TI
Non-Retirement	uop_queue_writes	Bit 0: FROM_TC_BUILD 1: FROM_TC_DELIVER 2: FROM_ROM	TS TS TS
Non-Retirement	resource_stall	Bit 5: SBFULL	TS
Non-Retirement	WC_Buffer	Bit 0: WCB_EVICTS 1: WCB_FULL_EVICT 2: WCB_HITM_EVICT	TI TI TI TI
At Retirement	instr_retired	Bit 0: NBOGUSNTAG 1: NBOGUSTAG 2: BOGUSNTAG 3: BOGUSTAG	TS TS TS TS
At Retirement	machine_clear	Bit 0: CLEAR 2: MOCLEAR 6: SMCCLEAR	TS TS TS
At Retirement	front_end_event	Bit 0: NBOGUS 1: BOGUS	TS TS
At Retirement	replay_event	Bit 0: NBOGUS 1: BOGUS	TS TS
At Retirement	execution_event	Bit 0: NONBOGUS0 1: NONBOGUS1	TS TS

Table 19-37. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		2: NONBOGUS2	TS
		3: NONBOGUS3	TS
		4: BOGUS0	TS
		5: BOGUS1	TS
		6: BOGUS2	TS
		7: BOGUS3	TS
At Retirement	x87_assist	Bit 0: FPSU	TS
		1: FPSO	TS
		2: POAO	TS
		3: POAU	TS
		4: PREA	TS
At Retirement	branch_retired	Bit 0: MMNP	TS
		1: MMNM	TS
		2: MMTP	TS
		3: MMTM	TS
At Retirement	mispred_branch_retired	Bit 0: NBOGUS	TS
At Retirement	uops_retired	Bit 0: NBOGUS	TS
		1: BOGUS	TS
At Retirement	instr_completed	Bit 0: NBOGUS	TS
		1: BOGUS	TS

19.19 PERFORMANCE MONITORING EVENTS FOR INTEL® PENTIUM® M PROCESSORS

The Pentium M processor's performance monitoring events are based on monitoring events for the P6 family of processors. All of these performance events are model specific for the Pentium M processor and are not available in this form in other processors. Table 19-38 lists the performance monitoring events that were added in the Pentium M processor.

Table 19-38. Performance Monitoring Events on Intel® Pentium® M Processors

Name	Hex Values	Descriptions
Power Management		
EMON_EST_TRANS	58H	Number of Enhanced Intel SpeedStep technology transitions: Mask = 00H - All transitions Mask = 02H - Only Frequency transitions
EMON_THERMAL_TRIP	59H	Duration/Occurrences in thermal trip; to count number of thermal trips: bit 22 in PerfEvtSel0/1 needs to be set to enable edge detect.
BPU		
BR_INST_EXEC	88H	Branch instructions that were executed (not necessarily retired).
BR_MISSP_EXEC	89H	Branch instructions executed that were mispredicted at execution.
BR_BAC_MISSP_EXEC	8AH	Branch instructions executed that were mispredicted at front end (BAC).
BR_CND_EXEC	8BH	Conditional branch instructions that were executed.
BR_CND_MISSP_EXEC	8CH	Conditional branch instructions executed that were mispredicted.
BR_IND_EXEC	8DH	Indirect branch instructions executed.
BR_IND_MISSP_EXEC	8EH	Indirect branch instructions executed that were mispredicted.
BR_RET_EXEC	8FH	Return branch instructions executed.
BR_RET_MISSP_EXEC	90H	Return branch instructions executed that were mispredicted at execution.
BR_RET_BAC_MISSP_EXEC	91H	Return branch instructions executed that were mispredicted at front end (BAC).
BR_CALL_EXEC	92H	CALL instruction executed.
BR_CALL_MISSP_EXEC	93H	CALL instruction executed and miss predicted.
BR_IND_CALL_EXEC	94H	Indirect CALL instructions executed.
Decoder		
EMON_SIMD_INSTR_RETIRED	CEH	Number of retired MMX instructions.
EMON_SYNCH_UOPS	D3H	Sync micro-ops
EMON_ESP_UOPS	D7H	Total number of micro-ops
EMON_FUSED_UOPS_RET	DAH	Number of retired fused micro-ops: Mask = 0 - Fused micro-ops Mask = 1 - Only load+Op micro-ops Mask = 2 - Only std+sta micro-ops
EMON_UNFUSION	DBH	Number of unfusion events in the ROB, happened on a FP exception to a fused μop.
Prefetcher		
EMON_PREF_RQSTS_UP	F0H	Number of upward prefetches issued.
EMON_PREF_RQSTS_DN	F8H	Number of downward prefetches issued.

A number of P6 family processor performance monitoring events are modified for the Pentium M processor. Table 19-39 lists the performance monitoring events that were changed in the Pentium M processor, and differ from performance monitoring events for the P6 family of processors.

Table 19-39. Performance Monitoring Events Modified on Intel® Pentium® M Processors

Name	Hex Values	Descriptions	
CPU_CLK_UNHALTED	79H	Number of cycles during which the processor is not halted, and not in a thermal trip.	
EMON_SSE_SSE2_INST_ RETIRED	D8H	Streaming SIMD Extensions Instructions Retired: Mask = 0 – SSE packed single and scalar single Mask = 1 – SSE scalar-single Mask = 2 – SSE2 packed-double Mask = 3 – SSE2 scalar-double	
EMON_SSE_SSE2_COMP_INST_ RETIRED	D9H	Computational SSE Instructions Retired: Mask = 0 – SSE packed single Mask = 1 – SSE Scalar-single Mask = 2 – SSE2 packed-double Mask = 3 – SSE2 scalar-double	
L2_LD	29H	L2 data loads	Mask[0] = 1 – count I state lines Mask[1] = 1 – count S state lines Mask[2] = 1 – count E state lines Mask[3] = 1 – count M state lines Mask[5:4]: 00H – Excluding hardware-prefetched lines 01H - Hardware-prefetched lines only 02H/03H – All (HW-prefetched lines and non HW -- Prefetched lines)
L2_LINES_IN	24H	L2 lines allocated	
L2_LINES_OUT	26H	L2 lines evicted	
L2_M_LINES_OUT	27H	Lw M-state lines evicted	

19.20 P6 FAMILY PROCESSOR PERFORMANCE MONITORING EVENTS

Table 19-40 lists the events that can be counted with the performance monitoring counters and read with the RDPMC instruction for the P6 family processors. The unit column gives the microarchitecture or bus unit that produces the event; the event number column gives the hexadecimal number identifying the event; the mnemonic event name column gives the name of the event; the unit mask column gives the unit mask required (if any); the description column describes the event; and the comments column gives additional information about the event.

All of these performance events are model specific for the P6 family processors and are not available in this form in the Pentium 4 processors or the Pentium processors. Some events (such as those added in later generations of the P6 family processors) are only available in specific processors in the P6 family. All performance event encodings not listed in Table 19-40 are reserved and their use will result in undefined counter results.

See the end of the table for notes related to certain entries in the table.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	43H	DATA_MEM_REFS	00H	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. The internal logic counts not only memory loads and stores, but also internal retries. 80-bit floating-point accesses are double counted, since they are decomposed into a 16-bit exponent load and a 64-bit mantissa load. Memory accesses are only counted when they are actually performed (such as a load that gets squashed because a previous cache miss is outstanding to the same address, and which finally gets performed, is only counted once). Does not include I/O accesses, or other nonmemory accesses.	
	45H	DCU_LINES_IN	00H	Total lines allocated in DCU.	
	46H	DCU_M_LINES_IN	00H	Number of M state lines allocated in DCU.	
	47H	DCU_M_LINES_ OUT	00H	Number of M state lines evicted from DCU. This includes evictions via snoop HITM, intervention or replacement.	
	48H	DCU_MISS_ OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding, incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. Uncacheable requests are excluded. Read-for-ownerships are counted, as well as line fills, invalidates, and stores.	An access that also misses the L2 is short-changed by 2 cycles (i.e., if counts N cycles, should be N+2 cycles). Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and noncacheable, including UC fetches.	
	81H	IFU_IFETCH_ MISS	00H	Number of instruction fetch misses All instruction fetches that do not hit the IFU (i.e., that produce memory requests). This includes UC accesses.	
	85H	ITLB_MISS	00H	Number of ITLB misses.	
	86H	IFU_MEM_STALL	00H	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults, and other minor stalls.	
	87H	ILD_STALL	00H	Number of cycles that the instruction length decoder is stalled.	
L2 Cache[1]	28H	L2_IFETCH	MESI 0FH	Number of L2 instruction fetches. This event indicates that a normal instruction fetch was received by the L2.	

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				The count includes only L2 cacheable instruction fetches; it does not include UC instruction fetches. It does not include ITLB miss accesses.	
	29H	L2_LD	MESI 0FH	Number of L2 data loads. This event indicates that a normal, unlocked, load memory access was received by the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It does include L2 cacheable TLB miss memory accesses.	
	2AH	L2_ST	MESI 0FH	Number of L2 data stores. This event indicates that a normal, unlocked, store memory access was received by the L2. it indicates that the DCU sent a read-for-ownership request to the L2. It also includes Invalid to Modified requests sent by the DCU to the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It includes TLB miss memory accesses.	
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_INM	00H	Number of modified lines allocated in the L2.	
	27H	L2_M_LINES_ OUTM	00H	Number of modified lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Total number of L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the L2 cache data bus was busy.	
	23H	L2_DBUS_BUSY_ RD	00H	Number of cycles during which the data bus was busy transferring read data from L2 to the processor.	
External Bus Logic (EBL)[2]	62H	BUS_DRDY_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY# is asserted. Utilization of the external system data bus during data transfers.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY#. Unit Mask = 20H counts in processor clocks when any agent is driving DRDY#.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	63H	BUS_LOCK_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK# is asserted on the external system bus.[3]	Always counts in processor clocks.
	60H	BUS_REQ_ OUTSTANDING	00H (Self)	Number of bus requests outstanding. This counter is incremented by the number of cacheable read bus requests outstanding in any given cycle.	Counts only DCU full-line cacheable reads, not RFOs, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of burst read transactions.	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of completed read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of completed write back transactions.	
	68H	BUS_TRAN_ IFETCH	00H (Self) 20H (Any)	Number of completed instruction fetch transactions.	
	69H	BUS_TRAN_INVA L	00H (Self) 20H (Any)	Number of completed invalidate transactions.	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	Number of completed partial write transactions.	
	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of completed partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of completed I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of completed deferred transactions.	

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of completed burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all completed bus transactions. Address bus utilization can be calculated knowing the minimum address bus occupancy. Includes special cycles, etc.	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	Number of completed memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR# pin.	
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow. ▪ If the PC bit is clear, the processor toggles the BPM*i* pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance monitoring counter events.
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
					▪ If the PC bit is clear, the processor toggles the BPM*i*pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance monitoring counter events.
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired. Excludes floating-point computational operations that cause traps or assists. Includes floating-point computational operations executed by the assist handler. Includes internal sub-operations for complex floating-point instructions like transcendentals. Excludes floating-point loads and stores.	Counter 0 only.
	10H	FP_COMP_OPS_ EXE	00H	Number of computational floating-point operations executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This number does not include the number of cycles, but the number of operations. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only. This event includes counts due to speculative execution.
	12H	MUL	00H	Number of multiplies. This count includes integer as well as FP multiplies and is speculative.	Counter 1 only.
	13H	DIV	00H	Number of divides. This count includes integer as well as FP divides and is speculative.	Counter 1 only.
	14H	CYCLES_DIV_ BUSY	00H	Number of cycles during which the divider is busy, and cannot accept new divides. This includes integer and FP divides, FPREM, FPSQRT, etc. and is speculative.	Counter 0 only.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Memory Ordering	03H	LD_BLOCKS	00H	Number of load operations delayed due to store buffer blocks. Includes counts caused by preceding stores whose addresses are unknown, preceding stores whose addresses are known but whose data is unknown, and preceding stores that conflicts with the load but which incompletely overlap the load.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles. Incremented every cycle the store buffer is draining. Draining is caused by serializing operations like CPUID, synchronizing operations like XCHG, interrupt acknowledgment, as well as other conditions (such as cache flushing).	
	05H	MISALIGN_ MEM_REF	00H	Number of misaligned data memory references. Incremented by 1 every cycle, during which either the processor's load or store pipeline dispatches a misaligned μop. Counting is performed if it is the first or second half, or if it is blocked, squashed, or missed. In this context, misaligned means crossing a 64-bit boundary.	MISALIGN_MEM_ REF is only an approximation to the true number of misaligned memory references. The value returned is roughly proportional to the number of misaligned memory accesses (the size of the problem).
	07H	EMON_KNI_PREF _DISPATCHED	 00H 01H 02H 03H	Number of Streaming SIMD extensions prefetch/weakly-ordered instructions dispatched (speculative prefetches are included in counting): 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
	4BH	EMON_KNI_PREF _MISS	 00H 01H 02H 03H	Number of prefetch/weakly-ordered instructions that miss all caches: 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
Instruction Decoding and Retirement	C0H	INST_RETIRED	00H	Number of instructions retired.	A hardware interrupt received during/after the last iteration of the REP STOS flow causes the counter to undercount by 1 instruction.
					An SMI received while executing a HLT instruction will cause the performance counter to not count the RSM instruction and undercount by 1.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	C2H	UOPS_RETIRED	00H	Number of μops retired.	
	D0H	INST_DECODED	00H	Number of instructions decoded.	
	D8H	EMON_KNI_INST_RETIRED	00H 01H	Number of Streaming SIMD extensions retired: 0: packed & scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
	D9H	EMON_KNI_COMP_INST_RET	00H 01H	Number of Streaming SIMD extensions computation instructions retired: 0: packed and scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
Interrupts	C8H	HW_INT_RX	00H	Number of hardware interrupts received.	
	C6H	CYCLES_INT_MASKED	00H	Number of processor cycles for which interrupts are disabled.	
	C7H	CYCLES_INT_PENDING_AND_MASKED	00H	Number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_RETIRED	00H	Number of branch instructions retired.	
	C5H	BR_MISS_PRED_RETIRED	00H	Number of mispredicted branches retired.	
	C9H	BR_TAKEN_RETIRED	00H	Number of taken branches retired.	
	CAH	BR_MISS_PRED_TAKEN_RET	00H	Number of taken mispredictions branches retired.	
	E0H	BR_INST_DECODED	00H	Number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Number of branches for which the BTB did not produce a prediction.	
	E4H	BR_BOGUS	00H	Number of bogus branches.	
	E6H	BACLEARS	00H	Number of times BACLEAR is asserted. This is the number of times that a static branch prediction was made, in which the branch decoder decided to make a branch prediction because the BTB did not.	
Stalls	A2H	RESOURCE_STALLS	00H	Incremented by 1 during every cycle for which there is a resource related stall. Includes register renaming buffer entries, memory buffer entries.	

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				Does not include stalls due to bus queue full, too many cache misses, etc. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls. This includes flag partial stalls.	
Segment Register Loads	06H	SEGMENT_REG_ LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX Unit	B0H	MMX_INSTR_ EXEC	00H	Number of MMX Instructions Executed.	Available in Intel Celeron, Pentium II and Pentium II Xeon processors only. Does not account for MOVQ and MOVD stores from register to memory.
	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX Saturating Instructions Executed.	Available in Pentium II and Pentium III processors only.
	B2H	MMX_UOPS_ EXEC	0FH	Number of MMX μops Executed.	Available in Pentium II and Pentium III processors only.
	B3H	MMX_INSTR_ TYPE_EXEC	01H 02H 04H 08H 10H 20H	MMX packed multiply instructions executed. MMX packed shift instructions executed. MMX pack operation instructions executed. MMX unpack operation instructions executed. MMX packed logical instructions executed. MMX packed arithmetic instructions executed.	Available in Pentium II and Pentium III processors only.
	CCH	FP_MMX_TRANS	00H 01H	Transitions from MMX instruction to floating-point instructions. Transitions from floating-point instructions to MMX instructions.	Available in Pentium II and Pentium III processors only.
	CDH	MMX_ASSIST	00H	Number of MMX Assists (that is, the number of EMMS instructions executed).	Available in Pentium II and Pentium III processors only.
	CEH	MMX_INSTR_RET	00H	Number of MMX Instructions Retired.	Available in Pentium II processors only.
Segment Register Renaming	D4H	SEG_RENAME_ STALLS		Number of Segment Register Renaming Stalls:	Available in Pentium II and Pentium III processors only.

Table 19-40. Events That Can Be Counted with the P6 Family Performance Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
			02H	Segment register ES	
			04H	Segment register DS	
			08H	Segment register FS	
			0FH	Segment register FS	
				Segment registers ES + DS + FS + GS	
	D5H	SEG_REG_ RENAMES		Number of Segment Register Renames:	Available in Pentium II and Pentium III processors only.
			01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register FS	
			0FH	Segment registers ES + DS + FS + GS	
	D6H	RET_SEG_ RENAMES	00H	Number of segment register rename events retired.	Available in Pentium II and Pentium III processors only.

NOTES:

1. Several L2 cache events, where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers. The lower 4 bits of the Unit Mask field are used in conjunction with L2 events to indicate the cache state or cache states involved.

   The P6 family processors identify cache states using the "MESI" protocol and consequently each bit in the Unit Mask field represents one of the four states: UMSK[3] = M (8H) state, UMSK[2] = E (4H) state, UMSK[1] = S (2H) state, and UMSK[0] = I (1H) state. UMSK[3:0] = MESI" (FH) should be used to collect data for all states; UMSK = 0H, for the applicable events, will result in nothing being counted.

2. All of the external bus logic (EBL) events, except where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers.

   Bit 5 of the UMSK field is used in conjunction with the EBL events to indicate whether the processor should count transactions that are self- generated (UMSK[5] = 0) or transactions that result from any processor on the bus (UMSK[5] = 1).

3. L2 cache locks, so it is possible to have a zero count.

19.21 PENTIUM PROCESSOR PERFORMANCE MONITORING EVENTS

Table 19-41 lists the events that can be counted with the performance monitoring counters for the Pentium processor. The Event Number column gives the hexadecimal code that identifies the event and that is entered in the ES0 or ES1 (event select) fields of the CESR MSR. The Mnemonic Event Name column gives the name of the event, and the Description and Comments columns give detailed descriptions of the events. Most events can be counted with either counter 0 or counter 1; however, some events can only be counted with only counter 0 or only counter 1 (as noted).

NOTE

The events in the table that are shaded are implemented only in the Pentium processor with MMX technology.

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters

Event Num.	Mnemonic Event Name	Description	Comments
00H	DATA_READ	Number of memory data reads (internal data cache hit and miss combined).	Split cycle reads are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
01H	DATA_WRITE	Number of memory data writes (internal data cache hit and miss combined); I/O not included.	Split cycle writes are counted individually. These events may occur at a maximum of two per clock. I/O is not included.
0H2	DATA_TLB_MISS	Number of misses to the data cache translation look-aside buffer.	
03H	DATA_READ_MISS	Number of memory read accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
04H	DATA WRITE MISS	Number of memory write accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
05H	WRITE_HIT_TO_ M-_OR_E-STATE_LINES	Number of write hits to exclusive or modified lines in the data cache.	These are the writes that may be held up if EWBE# is inactive. These events may occur a maximum of two per clock.
06H	DATA_CACHE_ LINES_ WRITTEN_BACK	Number of dirty lines (all) that are written back, regardless of the cause.	Replacements and internal and external snoops can all cause writeback and are counted.
07H	EXTERNAL_ SNOOPS	Number of accepted external snoops whether they hit in the code cache or data cache or neither.	Assertions of EADS# outside of the sampling interval are not counted, and no internal snoops are counted.
08H	EXTERNAL_DATA_ CACHE_SNOOP_ HITS	Number of external snoops to the data cache.	Snoop hits to a valid line in either the data cache, the data line fill buffer, or one of the write back buffers are all counted as hits.
09H	MEMORY ACCESSES IN BOTH PIPES	Number of data memory reads or writes that are paired in both pipes of the pipeline.	These accesses are not necessarily run in parallel due to cache misses, bank conflicts, etc.
0AH	BANK CONFLICTS	Number of actual bank conflicts.	
0BH	MISALIGNED DATA MEMORY OR I/O REFERENCES	Number of memory or I/O reads or writes that are misaligned.	A 2- or 4-byte access is misaligned when it crosses a 4-byte boundary; an 8-byte access is misaligned when it crosses an 8-byte boundary. Ten byte accesses are treated as two separate accesses of 8 and 2 bytes each.
0CH	CODE READ	Number of instruction reads; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0DH	CODE TLB MISS	Number of instruction reads that miss the code TLB whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0EH	CODE CACHE MISS	Number of instruction reads that miss the internal code cache; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
0FH	ANY SEGMENT REGISTER LOADED	Number of writes into any segment register in real or protected mode including the LDTR, GDTR, IDTR, and TR.	Segment loads are caused by explicit segment register load instructions, far control transfers, and task switches. Far control transfers and task switches causing a privilege level change will signal this event twice. Interrupts and exceptions may initiate a far control transfer.
10H	Reserved		
11H	Reserved		
12H	Branches	Number of taken and not taken branches, including: conditional branches, jumps, calls, returns, software interrupts, and interrupt returns.	Also counted as taken branches are serializing instructions, VERR and VERW instructions, some segment descriptor loads, hardware interrupts (including FLUSH#), and programmatic exceptions that invoke a trap or fault handler. The pipe is not necessarily flushed. The number of branches actually executed is measured, not the number of predicted branches.
13H	BTB_HITS	Number of BTB hits that occur.	Hits are counted only for those instructions that are actually executed.
14H	TAKEN_BRANCH_ OR_BTB_HIT	Number of taken branches or BTB hits that occur.	This event type is a logical OR of taken branches and BTB hits. It represents an event that may cause a hit in the BTB. Specifically, it is either a candidate for a space in the BTB or it is already in the BTB.
15H	PIPELINE FLUSHES	Number of pipeline flushes that occur Pipeline flushes are caused by BTB misses on taken branches, mispredictions, exceptions, interrupts, and some segment descriptor loads.	The counter will not be incremented for serializing instructions (serializing instructions cause the prefetch queue to be flushed but will not trigger the Pipeline Flushed event counter) and software interrupts (software interrupts do not flush the pipeline).
16H	INSTRUCTIONS_ EXECUTED	Number of instructions executed (up to two per clock).	Invocations of a fault handler are considered instructions. All hardware and software interrupts and exceptions will also cause the count to be incremented. Repeat prefixed string instructions will only increment this counter once despite the fact that the repeat loop executes the same instruction multiple times until the loop criteria is satisfied. This applies to all the Repeat string instruction prefixes (i.e., REP, REPE, REPZ, REPNE, and REPNZ). This counter will also only increment once per each HLT instruction executed regardless of how many cycles the processor remains in the HALT state.
17H	INSTRUCTIONS_ EXECUTED_ V PIPE	Number of instructions executed in the V_pipe. The event indicates the number of instructions that were paired.	This event is the same as the 16H event except it only counts the number of instructions actually executed in the V-pipe.
18H	BUS_CYCLE_ DURATION	Number of clocks while a bus cycle is in progress. This event measures bus use.	The count includes HLDA, AHOLD, and BOFF# clocks.
19H	WRITE_BUFFER_ FULL_STALL_ DURATION	Number of clocks while the pipeline is stalled due to full write buffers.	Full write buffers stall data memory read misses, data memory write misses, and data memory write hits to S-state lines. Stalls on I/O accesses are not included.

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
1AH	WAITING_FOR_ DATA_MEMORY_ READ_STALL_ DURATION	Number of clocks while the pipeline is stalled while waiting for data memory reads.	Data TLB Miss processing is also included in the count. The pipeline stalls while a data memory read is in progress including attempts to read that are not bypassed while a line is being filled.
1BH	STALL ON WRITE TO AN E- OR M-STATE LINE	Number of stalls on writes to E- or M-state lines.	
1CH	LOCKED BUS CYCLE	Number of locked bus cycles that occur as the result of the LOCK prefix or LOCK instruction, page-table updates, and descriptor table updates.	Only the read portion of the locked read-modify-write is counted. Split locked cycles (SCYC active) count as two separate accesses. Cycles restarted due to BOFF# are not re-counted.
1DH	I/O READ OR WRITE CYCLE	Number of bus cycles directed to I/O space.	Misaligned I/O accesses will generate two bus cycles. Bus cycles restarted due to BOFF# are not re-counted.
1EH	NONCACHEABLE_ MEMORY_READS	Number of noncacheable instruction or data memory read bus cycles. The count includes read cycles caused by TLB misses, but does not include read cycles to I/O space.	Cycles restarted due to BOFF# are not re-counted.
1FH	PIPELINE_AGI_ STALLS	Number of address generation interlock (AGI) stalls. An AGI occurring in both the U- and V-pipelines in the same clock signals this event twice.	An AGI occurs when the instruction in the execute stage of either of U- or V-pipelines is writing to either the index or base address register of an instruction in the D2 (address generation) stage of either the U- or V- pipelines.
20H	Reserved		
21H	Reserved		
22H	FLOPS	Number of floating-point operations that occur.	Number of floating-point adds, subtracts, multiplies, divides, remainders, and square roots are counted. The transcendental instructions consist of multiple adds and multiplies and will signal this event multiple times. Instructions generating the divide-by-zero, negative square root, special operand, or stack exceptions will not be counted. Instructions generating all other floating-point exceptions will be counted. The integer multiply instructions and other instructions which use the x87 FPU will be counted.
23H	BREAKPOINT MATCH ON DR0 REGISTER	Number of matches on register DR0 breakpoint.	The counters is incremented regardless if the breakpoints are enabled or not. However, if breakpoints are not enabled, code breakpoint matches will not be checked for instructions executed in the V-pipe and will not cause this counter to be incremented. (They are checked on instruction executed in the U-pipe only when breakpoints are not enabled.) These events correspond to the signals driven on the BP[3:0] pins. Refer to Chapter 17, "Debug, Branch Profile, TSC, and Resource Monitoring Features" for more information.
24H	BREAKPOINT MATCH ON DR1 REGISTER	Number of matches on register DR1 breakpoint.	See comment for 23H event.

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
25H	BREAKPOINT MATCH ON DR2 REGISTER	Number of matches on register DR2 breakpoint.	See comment for 23H event.
26H	BREAKPOINT MATCH ON DR3 REGISTER	Number of matches on register DR3 breakpoint.	See comment for 23H event.
27H	HARDWARE INTERRUPTS	Number of taken INTR and NMI interrupts.	
28H	DATA_READ_OR_ WRITE	Number of memory data reads and/or writes (internal data cache hit and miss combined).	Split cycle reads and writes are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
29H	DATA_READ_MISS OR_WRITE MISS	Number of memory read and/or write accesses that miss the internal data cache, whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
2AH	BUS_OWNERSHIP_ LATENCY (Counter 0)	The time from LRM bus ownership request to bus ownership granted (that is, the time from the earlier of a PBREQ (0), PHITM# or HITM# assertion to a PBGNT assertion)	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2AH	BUS OWNERSHIP TRANSFERS (Counter 1)	The number of buss ownership transfers (that is, the number of PBREQ (0) assertions	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ U-PIPE (Counter 0)	Number of MMX instructions executed in the U-pipe	
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ V-PIPE (Counter 1)	Number of MMX instructions executed in the V-pipe	
2CH	CACHE_M-STATE_LINE_ SHARING (Counter 0)	Number of times a processor identified a hit to a modified line due to a memory access in the other processor (PHITM (0))	If the average memory latencies of the system are known, this event enables the user to count the Write Backs on PHITM(0) penalty and the Latency on Hit Modified(I) penalty.
2CH	CACHE_LINE_ SHARING (Counter 1)	Number of shared data lines in the L1 cache (PHIT (0))	
2DH	EMMS_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of EMMS instructions executed	

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
2DH	TRANSITIONS_ BETWEEN_MMX_ AND_FP_ INSTRUCTIONS (Counter 1)	Number of transitions between MMX and floating-point instructions or vice versa An even count indicates the processor is in MMX state. an odd count indicates it is in FP state.	This event counts the first floating-point instruction following an MMX instruction or first MMX instruction following a floating-point instruction. The count may be used to estimate the penalty in transitions between floating-point state and MMX state.
2EH	BUS_UTILIZATION_ DUE_TO_ PROCESSOR_ ACTIVITY (Counter 0)	Number of clocks the bus is busy due to the processor's own activity (the bus activity that is caused by the processor)	
2EH	WRITES_TO_ NONCACHEABLE_ MEMORY (Counter 1)	Number of write accesses to noncacheable memory	The count includes write cycles caused by TLB misses and I/O write cycles. Cycles restarted due to BOFF# are not re-counted.
2FH	SATURATING_ MMX_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of saturating MMX instructions executed, independently of whether they actually saturated.	
2FH	SATURATIONS_ PERFORMED (Counter 1)	Number of MMX instructions that used saturating arithmetic when at least one of its results actually saturated	If an MMX instruction operating on 4 doublewords saturated in three out of the four results, the counter will be incremented by one only.
30H	NUMBER_OF_ CYCLES_NOT_IN_ HALT_STATE (Counter 0)	Number of cycles the processor is not idle due to HLT instruction	This event will enable the user to calculate "net CPI". Note that during the time that the processor is executing the HLT instruction, the Time-Stamp Counter is not disabled. Since this event is controlled by the Counter Controls CC0, CC1 it can be used to calculate the CPI at CPL=3, which the TSC cannot provide.
30H	DATA_CACHE_ TLB_MISS_ STALL_DURATION (Counter 1)	Number of clocks the pipeline is stalled due to a data cache translation look-aside buffer (TLB) miss	
31H	MMX_ INSTRUCTION_ DATA_READS (Counter 0)	Number of MMX instruction data reads	
31H	MMX_ INSTRUCTION_ DATA_READ_ MISSES (Counter 1)	Number of MMX instruction data read misses	
32H	FLOATING_POINT_S TALLS_DURATION (Counter 0)	Number of clocks while pipe is stalled due to a floating-point freeze	
32H	TAKEN_BRANCHES (Counter 1)	Number of taken branches	

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
33H	D1_STARVATION_ AND_FIFO_IS_ EMPTY (Counter 0)	Number of times D1 stage cannot issue ANY instructions since the FIFO buffer is empty	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer.
33H	D1_STARVATION_ AND_ONLY_ONE_ INSTRUCTION_IN_ FIFO (Counter 1)	Number of times the D1 stage issues a single instruction (since the FIFO buffer had just one instruction ready)	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer. When combined with the previously defined events, Instruction Executed (16H) and Instruction Executed in the V-pipe (17H), this event enables the user to calculate the numbers of time pairing rules prevented issuing of two instructions.
34H	MMX_ INSTRUCTION_ DATA_WRITES (Counter 0)	Number of data writes caused by MMX instructions	
34H	MMX_ INSTRUCTION_ DATA_WRITE_ MISSES (Counter 1)	Number of data write misses caused by MMX instructions	
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS (Counter 0)	Number of pipeline flushes due to wrong branch predictions resolved in either the E-stage or the WB-stage	The count includes any pipeline flush due to a branch that the pipeline did not follow correctly. It includes cases where a branch was not in the BTB, cases where a branch was in the BTB but was mispredicted, and cases where a branch was correctly predicted but to the wrong address. Branches are resolved in either the Execute stage (E-stage) or the Writeback stage (WB-stage). In the later case, the misprediction penalty is larger by one clock. The difference between the 35H event count in counter 0 and counter 1 is the number of E-stage resolved branches.
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS_ RESOLVED_IN_ WB-STAGE (Counter 1)	Number of pipeline flushes due to wrong branch predictions resolved in the WB-stage	See note for event 35H (Counter 0).
36H	MISALIGNED_ DATA_MEMORY_ REFERENCE_ON_ MMX_ INSTRUCTIONS (Counter 0)	Number of misaligned data memory references when executing MMX instructions	
36H	PIPELINE_ ISTALL_FOR_MMX_ INSTRUCTION_ DATA_MEMORY_ READS (Counter 1)	Number clocks during pipeline stalls caused by waits form MMX instruction data memory reads	T3:

Table 19-41. Events That Can Be Counted with Pentium Processor Performance Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
37H	MISPREDICTED_ OR_ UNPREDICTED_ RETURNS (Counter 1)	Number of returns predicted incorrectly or not predicted at all	The count is the difference between the total number of executed returns and the number of returns that were correctly predicted. Only RET instructions are counted (for example, IRET instructions are not counted).
37H	PREDICTED_ RETURNS (Counter 1)	Number of predicted returns (whether they are predicted correctly and incorrectly	Only RET instructions are counted (for example, IRET instructions are not counted).
38H	MMX_MULTIPLY_ UNIT_INTERLOCK (Counter 0)	Number of clocks the pipe is stalled since the destination of previous MMX multiply instruction is not ready yet	The counter will not be incremented if there is another cause for a stall. For each occurrence of a multiply interlock, this event will be counted twice (if the stalled instruction comes on the next clock after the multiply) or by once (if the stalled instruction comes two clocks after the multiply).
38H	MOVD/MOVQ_ STORE_STALL_ DUE_TO_ PREVIOUS_MMX_ OPERATION (Counter 1)	Number of clocks a MOVD/MOVQ instruction store is stalled in D2 stage due to a previous MMX operation with a destination to be used in the store instruction.	
39H	RETURNS (Counter 0)	Number or returns executed.	Only RET instructions are counted; IRET instructions are not counted. Any exception taken on a RET instruction and any interrupt recognized by the processor on the instruction boundary prior to the execution of the RET instruction will also cause this counter to be incremented.
39H	Reserved		
3AH	BTB_FALSE_ ENTRIES (Counter 0)	Number of false entries in the Branch Target Buffer	False entries are causes for misprediction other than a wrong prediction.
3AH	BTB_MISS_ PREDICTION_ON_ NOT-TAKEN_ BRANCH (Counter 1)	Number of times the BTB predicted a not-taken branch as taken	
3BH	FULL_WRITE_ BUFFER_STALL_ DURATION_ WHILE_ EXECUTING_MMX_I NSTRUCTIONS (Counter 0)	Number of clocks while the pipeline is stalled due to full write buffers while executing MMX instructions	
3BH	STALL_ON_MMX_ INSTRUCTION_ WRITE_TO E-_OR_ M-STATE_LINE (Counter 1)	Number of clocks during stalls on MMX instructions writing to E- or M-state lines	

20.Updates to Chapter 24, Volume 3B

Change bars show changes to Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Various updates related to VMX and Intel Processor Trace interactions.

CHAPTER 24
VIRTUAL MACHINE CONTROL STRUCTURES

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.

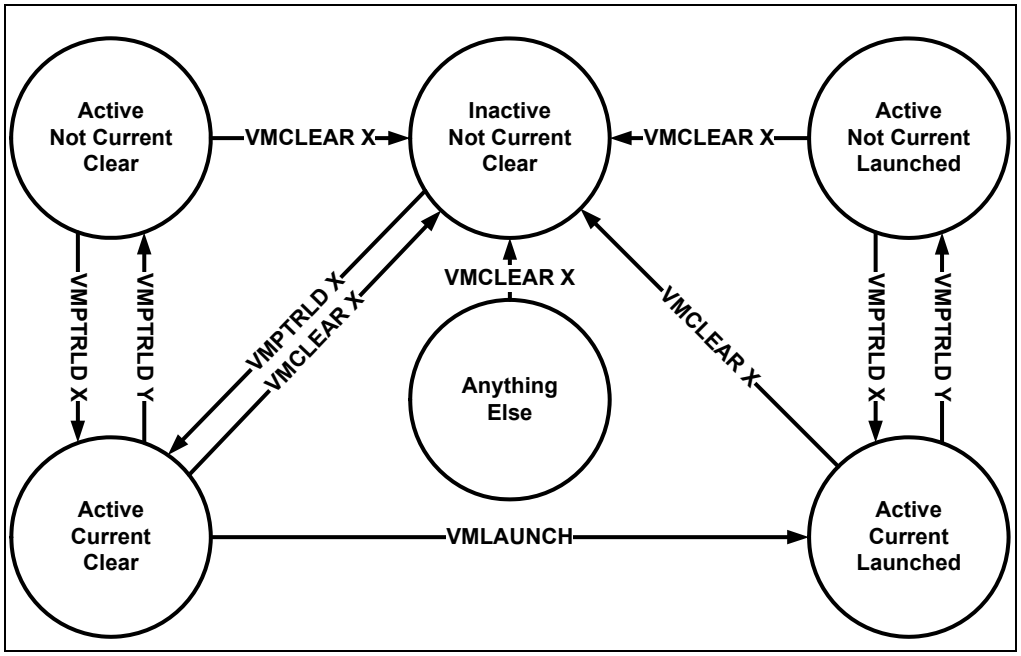


Figure 24-1. States of VMCS X

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)

The first 4 bytes of the VMCS region contain the **VMCS revision identifier** at bits 30:0.[2] Processors that maintain VMCS data in different formats (see below) use different VMCS revision identifiers. These identifiers enable soft-

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).

ware to avoid using a VMCS region formatted for one processor on a processor that uses a different format.[1] 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 type[2]. 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.[3]

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are sometimes referred to collectively as VMX controls.

---

2. 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.

1. Logical processors that use the same VMCS revision identifier use the same size for VMCS regions.

2. 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.

3. 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. As noted earlier, processor state is loaded from these fields on every VM entry (see Section 26.3.2) and stored into these fields on every VM exit (see Section 27.3).

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]

- 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.
- 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:[2]

  - 0: **Active**. The logical processor is executing instructions normally.

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.

— 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**[1] 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Execution of STI with RFLAGS.IF = 0 blocks interrupts (and, optionally, other events) for one instruction after its execution. Setting this bit indicates that this blocking is in effect.
1	Blocking by MOV SS	See the "MOV—Move a Value from the Stack" from Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and "POP—Pop a Value from the Stack" from Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and Section 6.8.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Execution of a MOV to SS or a POP to SS blocks interrupts for one instruction after its execution. In addition, certain debug exceptions are inhibited between a MOV to SS or a POP to SS and a subsequent instruction. Setting this bit indicates that the blocking of all these events is in effect. 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 interrupts (SMIs) are disabled while the processor is in system-management mode (SMM). Setting this bit indicates that blocking of SMIs is in effect.
3	Blocking by NMI	See Section 6.7.1 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* and Section 34.8. 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	A VM exit saves this bit as 1 to indicate that the VM exit was incident to enclave mode.
31:5	Reserved	VM entry will fail if these bits are not 0. See Section 26.3.1.5.

- **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.[2] This field contains information about such exceptions. This field is described in Table 24-4.

2. 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.

1. A triple fault occurs when a logical processor encounters an exception while attempting to deliver a double fault.

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.
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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).[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 26.6.4.
- **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.)

---

2. 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*.

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 28.2.5.

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 support

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/INSW/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 unpaged 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 25.5.5.
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 28.2.5.
18	EPT-violation #VE	If this control is 1, EPT violations may cause virtualization exceptions (#VE) instead of VM exits. See Section 25.5.6.
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.
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).

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS2 (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 27 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 accesses 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).

  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.

1. If the local APIC does not support x2APIC mode, it is always in xAPIC mode.

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 toC0001FFFH. 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 toC0001FFFH. 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 28.2.6): 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 28.2.4)[2]
11:7	Reserved
N–1:12	Bits N–1:12 of the physical address of the 4-KByte aligned EPT PML4 table[3]
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. 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 25.5.5 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 25.5.5.3.

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 25.5.5.3).

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 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 28.2.5.

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.

24.6.18 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 25.5.6.2.

- **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 25.5.5.3).

24.6.19 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).

24.7 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 24-10 lists the controls supported. See Chapter 27 for complete details of how these controls affect VM exits.

Table 24-10. 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: ▪ 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).

NOTES:

1. Since Intel 64 architecture specifies that IA32_EFER.LMA is always set to the logical-AND of CR0.PG and IA32_EFER.LME, and since CR0.PG is always 1 in VMX operation, IA32_EFER.LMA is always identical to IA32_EFER.LME in VMX 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_CTLS and IA32_VMX_TRUE_EXIT_CTLS (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_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_EXIT_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.

24.7.2 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 bytes.[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 24-11. If the VM-exit MSR-store count is not zero, the address must be 16-byte aligned.

Table 24-11. 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 bytes. 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 24-11). 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.

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).

24.8.1 VM-Entry Controls

The **VM-entry controls** constitute a 32-bit vector that governs the basic operation of VM entries. Table 24-12 lists the controls supported. See Chapter 24 for how these controls affect VM entries.

Table 24-12. 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).

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).

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_CTLS and IA32_VMX_TRUE_ENTRY_CTLS (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).

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 bytes. Otherwise, unpredictable processor behavior (including a machine check) may result during VM entry.[1]

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).

- **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 24-11. If the VM-entry MSR-load count is not zero, the address must be 16-byte aligned.

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 24-13 describes the field.

Table 24-13. 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 4: Software interrupt 5: Privileged software exception 6: Software exception 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 breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); it should use the type **software exception** for #BP and #OF. The type **other event** is used for injection of events that are not delivered through the IDT.
  - 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 26.5 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.

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 24-14.

Table 24-14. Format of Exit Reason

Bit Position(s)	Contents
15:0	Basic exit reason
26:16	Reserved (cleared to 0)
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	Reserved (cleared to 0)
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 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 26.7), 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; 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.
  - — 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.

1. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

- **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, BOUND, and UD); 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 24-15 describes this field.

Table 24-15. 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 – 5: Not used 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
12	NMI unblocking due to IRET
30:13	Reserved (cleared to 0)
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:

- **IDT-vectoring information** (32 bits). This field receives basic information associated with the event that was being delivered when the VM exit occurred. Table 24-16 describes this field.

1. This includes cases in which the event delivery was caused by event injection as part of VM entry; see Section 26.5.1.2.

Table 24-16. 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)
12	Undefined
30:13	Reserved (cleared to 0)
31	Valid

- **IDT-vectoring error code** (32 bits). For VM exits the occur during delivery of hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

See Section 27.2.3 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 27.2.4 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 27.2.4 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).

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.

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).

24.10 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 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.

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.

- 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 27 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 24-17.

Table 24-17. 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 unitize 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).

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:

- The VMXON pointer must be 4-KByte aligned (bits 11:0 must be zero).
- The VMXON pointer must not set any bits beyond the processor's physical-address width.[2,3]

---

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).
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, the VMXON pointer must not set any bits in the range 63:32; see Appendix A.1.

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).

21. 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.

Change to chapter: Footnote update in Section 28.3.3.4 "Guidelines for Use of the INVEPT Instruction".

CHAPTER 28
VMX SUPPORT FOR ADDRESS TRANSLATION

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 28.2.6 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 from 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 28.2.4). 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 from 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 28.2.6)
6	Ignore PAT memory type for this 1-GByte page (see Section 28.2.6)
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 28.2.4). 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 28.2.4). 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)
62:52	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 25.5.6.1). 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 from 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 28.2.4). 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 from 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 28.2.6)
6	Ignore PAT memory type for this 2-MByte page (see Section 28.2.6)
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 28.2.4). 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 28.2.4). 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)
62:52	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 25.5.6.1). 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-struc-

ture 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 25.5.6.1. 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 from 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 28.2.4). 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 28.2.6.

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 28.2.5).

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 from 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 28.2.6)
6	Ignore PAT memory type for this 4-KByte page (see Section 28.2.6)
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 28.2.4). 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 28.2.4). 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)
62:52	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 25.5.6.1). 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.

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 25.5.6.1.

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).[1]
  - 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.[2] 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.

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.

---

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.

1. 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.

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.

63	62	61	60	59	58	57	56	55	54	53	52	51	M[1]	M-1	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved														Address of EPT PML4 table																						Rsvd.							A/D	EPT PWL-1		EPT PS MT			**EPTP[2]**
Ignored													Rsvd.	Address of EPT page-directory-pointer table																					Ign.	XU[3]	Ign.	A	Reserved						X[4]	W	R	**PML4E: present[5]**
SVE[6]	Ignored																																													**0**	**0**	**0**	**PML4E: not present**
SVE	Ignored													Rsvd.	Physical address of 1GB page																					Reserved										Ign.	XU	D	A	**1**	IPAT	EPT MT	X	W	R	**PDPTE: 1GB page**
Ignored														Rsvd.	Address of EPT page directory																															Ign.	XU	Ign.	A	**0**	Rsvd.		X	W	R	**PDPTE: page directory**
SVE	Ignored																																														**0**	**0**	**0**	**PDTPE: not present**
SVE	Ignored													Rsvd.	Physical address of 2MB page																					Reserved										Ign.	XU	D	A	**1**	IPAT	EPT MT	X	W	R	**PDE: 2MB page**
Ignored														Rsvd.	Address of EPT page table																															Ign.	XU	Ign.	A	**0**	Rsvd.		X	W	R	**PDE: page table**
SVE	Ignored																																														**0**	**0**	**0**	**PDE: not present**
SVE	Ignored													Rsvd.	Physical address of 4KB page																															Ign.	XU	D	A	Ign	IPAT	EPT MT	X	W	R	**PTE: 4KB page**
SVE	Ignored																																														**0**	**0**	**0**	**PTE: not present**

Figure 28-1. Formats of EPTP and EPT Paging-Structure Entries

NOTES:

1. M is an abbreviation for MAXPHYADDR.
2. See Section 24.6.11 for details of the EPTP.
3. Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 0, this bit is ignored.
4. 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.
5. 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.
6. Suppress #VE. If the "EPT-violation #VE" VM-execution control is 0, this bit is ignored.

- The access is a data write, 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.)

- 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.

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):
   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

1. This is a simplification of the more detailed description given in Section 28.2.2.

guest-physical address in the appropriate PDPTE register), then accessing an entry in another guest paging structure referenced by the guest-physical address in the first guest paging-structure entry, etc. Each guest-physical address is itself translated using EPT and may cause an EPT-induced VM exit. The following items detail how page faults and EPT-induced VM exits are recognized during this iterative process:

1. An attempt is made to access a guest paging-structure entry with a guest-physical address (initially, the address in CR3 or PDPTE register).
   a. If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
   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 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 PDPTE registers for PAE paging by the MOV to CR instruction; see Section 4.4.1. Those loads of guest PDPTEs 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.

28.2.5 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 28.2.4). 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 28.2.4. 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).

28.2.6 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 28.2.6.1 explains how the memory type is determined for accesses to the EPT paging structures. Section 28.2.6.2 explains how the memory type is determined for an access using a guest-physical address that is translated using EPT.

28.2.6.1 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.

28.2.6.2 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) 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

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 PDPTEs is WB.

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.

28.3.2 Creating and Using Cached Translation Information

The following items detail the creation of the mappings described in the previous section:[4]

- 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:

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.
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*.

— 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.

28.3.3 Invalidating Cached Translation Information

Software modifications of paging structures (including EPT paging structures) 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.[3] They are required to do so only for the current VPID (but, for combined mappings, all EP4TAs). Linear

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_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.
3. 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.

mappings for the current VPID are invalidated even if EPT is in use.[1] Combined mappings for the current VPID are invalidated even if EPT is not in use.[2]

- 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.
- 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.

1. 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.
2. 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.

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 PDPTE 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 PDPTE 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 28.2.6.)
  - For the **last** EPT paging-structure entry used to translate a guest-physical address (an EPT PDPTE 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.
- 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.[2] 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.

---

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.
2. 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.

- 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 shootdown." A discussion of TLB shootdown 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*.

22.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: Various updates related to VMX and Intel Processor Trace interactions. Various minor corrections throughout chapter.

CHAPTER 35
INTEL® PROCESSOR TRACE

35.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The initial implementations of Intel PT offer **control flow tracing**, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

35.1.1 Features and Capabilities

Intel PT's control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

Intel PT can also be configured to log software-generated packets using PTWRITE, and packets describing processor power management events.

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32_RTIT_*. The capability provided by these configuration MSRs are enumerated by CPUID, see Section 35.3. Details of the MSRs for configuring Intel PT are described in Section 35.2.7.

35.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following categories of packets (for more details on the packets, see Section 35.4):

- Packets about basic information on program execution; these include:
  - Packet Stream Boundary (**PSB**) packets: PSB packets act as 'heartbeats' that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
  - Paging Information Packet (**PIP**): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
  - Time-Stamp Counter (**TSC**) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
  - Core Bus Ratio (**CBR**) packets: CBR packets contain the core:bus clock ratio.

  - Overflow (**OVF**) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.
- Packets about control flow information:
  - Taken Not-Taken (**TNT**) packets: TNT packets track the "direction" of direct conditional branches (taken or not taken).
  - Target IP (**TIP**) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 35.4.2.2.
  - Flow Update Packets (**FUP**): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
  - **MODE** packets: These packets provide the decoder with important processor execution information so that it can properly interpret the dis-assembled binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).
- Packets inserted by software:
  - PTWRITE (PTW) packets: includes the value of the operand passed to the PTWRITE instruction (see "PTWRITE - Write Data to a Processor Trace Packet" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).
- Packets about processor power management events:
  - MWAIT packets: Indicate successful completion of an MWAIT operation to a C-state deeper than C0.0.
  - Power State Entry (PWRE) packets: Indicate entry to a C-state deeper than C0.0.
  - Power State Exit (PWRX) packets: Indicate exit from a C-state deeper than C0.0, returning to C0.
  - Execution Stopped (EXSTOP) packets: Indicate that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling.

35.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

35.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). There are three categories of COFI:

- Direct transfer COFI.
- Indirect transfer COFI.
- Far transfer COFI.

The following subsections describe the COFI events that result in trace packet generation. Table 35-1 lists branch instruction by COFI types. For detailed description of specific instructions, see *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

35.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two 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 35-23 indicates exactly which IP will be included in the FUP generated by a far transfer.

35.2.2 Software Trace Instrumentation with PTWRITE

PTWRITE provides a mechanism by which software can instrument the Intel PT trace. PTWRITE is a ring3-accessible instruction that can be passed 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 35-40 "PTW Packet Definition"), inserted at the time of PTWRITE retirement, assuming PTWRITE is enabled and all other filtering conditions are met. Decode and analysis software will then be able to determine the meaning of the PTWRITE packet based on the IP of the associated PTWRITE instruction.

PTWRITE is enabled via IA32_RTIT_CTL.PTWEn[12] (see Table 35-6). Optionally, the user can use IA32_RTIT_CTL.FUPonPTW[5] to enable PTW packets to be followed by FUP packets containing the IP of the associated PTWRITE instruction.

35.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

- Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.
- The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.
- The last and deepest core C-state achieved during a sleep session.
- The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32_RTIT_CTL.PwrEvtEn[4].

35.2.4 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

35.2.4.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 35.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 35.2.5.1 for details.

35.2.4.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 35.3.5, “Context Switch Consideration”).

To trace for only a single CR3 value, software can write that value to the IA32_RTIT_CR3_MATCH MSR, and set IA32_RTIT_CTL.CR3Filter. When CR3 value does not match IA32_RTIT_CR3_MATCH and IA32_RTIT_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 35.2.5.3 for details. When CR3 does match IA32_RTIT_CR3_MATCH (or when IA32_RTIT_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32_RTIT_CTL.OS = 1). If not, no PIP packets will be seen.

35.2.4.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1. Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn_CFG fields in the IA32_RTIT_CTL MSR (Section 35.2.7.2), where the digit 'n' is a zero-based number that selects which address range is being configured. Each ADDRn_CFG field configures the use of the register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B (Section 35.2.7.5). IA32_RTIT_ADDRn_A defines the base and IA32_RTIT_ADDRn_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32_RTIT_ADDRn_A. IA32_RTIT_ADDRn_B]. There can be multiple such ranges, software can query CPUID (Section 35.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn_CFG is set to enable IP filtering (see Section 35.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 35.2.5.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 35.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 35.3.1).

Note that some packets, such as MTC (Section 35.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 35.4.1.

TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32_RTIT_CTL.ADDRn_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 35.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.

To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32_RTIT_CTL.TraceEn before the IA32_RTIT_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32_RTIT_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 35.2.6.2.

IP Filtering Example

The following table gives an example of IP filtering behavior. Assume that IA32_RTIT_ADDRn_A = the IP of RangeBase, and that IA32_RTIT_ADDRn_B = the IP of RangeLimit, while IA32_RTIT_CTL.ADDRn_CFG = 0x1 (enable ADDRn range as a FilterEn range).

Table 35-2. IP Filtering Packet Example

Code Flow	Packets

`Bar:`
`  jmp RangeBase // jump into filter range`
`RangeBase:`
`  jcc Foo // not taken`
`  add eax, 1`
`Foo:`
`  jmp RangeLimit+1 // jump out of filter range`
`RangeLimit:`
`  nop`
`  jcc Bar`
	TIP.PGE(RangeBase)
	TNT(0)
	TIP.PGD(RangeLimit+1)

IP Filtering and TraceStop

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.

35.2.5 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (**PacketEn**) is set. PacketEn is an internal state maintained in hardware in

response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

35.2.5.1 Packet Enable (PacketEn)

When PacketEn is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. PacketEn is composed of other states according to this relationship:

```
PacketEn ← TriggerEn AND ContextEn AND FilterEn AND BranchEn
```

These constituent controls are detailed in the following subsections.

PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and bookkeeping packets) may still be sent. See Section 35.2.6 for details of PacketEn and packet generation.

Note that, on processors that do not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0), FilterEn is treated as always set.

35.2.5.2 Trigger Enable (TriggerEn)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32_RTIT_CTL.TraceEn is set, and cleared by any of the following conditions:

- TraceEn is cleared by software.
- A TraceStop condition is encountered and IA32_RTIT_STATUS.Stopped is set.
- IA32_RTIT_STATUS.Error is set due to an operational error (see Section 35.3.9).

Software can discover the current TriggerEn value by reading the IA32_RTIT_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

35.2.5.3 Context Enable (ContextEn)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32_RTIT_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32_RTIT_STATUS.ContextEn bit. ContextEn is defined as follows:

```
ContextEn = !((IA32_RTIT_CTL.OS = 0 AND CPL = 0) OR
(IA32_RTIT_CTL.USER = 0 AND CPL > 0) OR (IS_IN_A_PRODUCTION_ENCLAVE[1]) OR
(IA32_RTIT_CTL.CR3Filter = 1 AND IA32_RTIT_CR3_MATCH does not match CR3)
```

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets (TNT, FUP, TIP.*, MODE.*) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as MTC and PSB (see Section 35.4.2.16 and Section 35.4.2.17), may still be generated while ContextEn is 0. For details of which packets are generated only when ContextEn is set, see Section 35.4.1.

The processor does not update ContextEn when TriggerEn = 0.

The value of ContextEn will toggle only when TriggerEn = 1.

35.2.5.4 Branch Enable (BranchEn)

This value is based purely on the IA32_RTIT_CTL.BranchEn value. If **BranchEn** is not set, then relevant COFI packets (TNT, TIP*, FUP, MODE.*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well

1. Trace packets generation is disabled in a production enclave, see Section 35.2.8.5. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

35.2.5.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32_RTIT_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 35.2.4.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 35.4.1.

After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32_RTIT_CTL.ADDRn_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

35.2.6 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32_RTIT_CTL (see Section 35.2.7.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-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.MaskOrTableOffset > 0.

- Illegal Output Offset

  IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than the mask value (IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset).

Also note that errors can be signaled due to trace packet output overlapping with restricted memory, see Section 35.2.6.4.

35.2.6.2 Table of Physical Addresses (ToPA)

When IA32_RTIT_CTL.ToPA is set and IA32_RTIT_CTL.FabricEn is clear, the ToPA output mechanism is utilized. The ToPA mechanism uses a linked list of tables; see Figure 35-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- **proc_trace_table_base.**
  This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32_RTIT_OUTPUT_BASE MSR. While tracing is enabled, the processor updates the IA32_RTIT_OUTPUT_BASE MSR with changes to proc_trace_table_base, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_base.

- **proc_trace_table_offset.**
  This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads this value from bits 31:7 (MaskOrTableOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset with changes to proc_trace_table_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_offset.

- **proc_trace_output_offset.**
  This a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 (OutputOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates

IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset with changes to proc_trace_output_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_output_offset.

Figure 35-1 provides an illustration (not to scale) of the table and associated pointers.

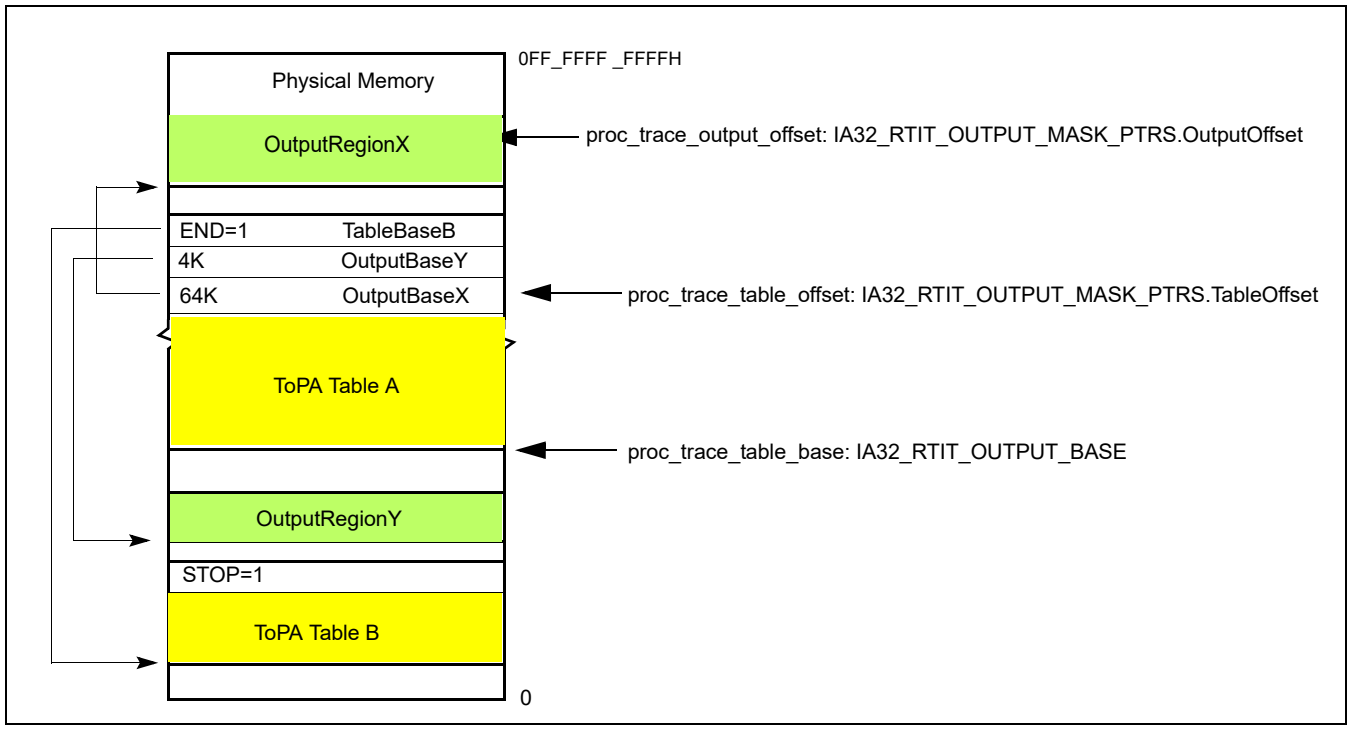


Figure 35-1. ToPA Memory Illustration

With the ToPA mechanism, the processor writes packets to the current output region (identified by proc_trace_table_base and the proc_trace_table_offset). The offset within that region to which the next byte will be written is identified by proc_trace_output_offset. When that region is filled with packet output (thus proc_trace_output_offset = RegionSize–1), proc_trace_table_offset is moved to the next ToPA entry, proc_trace_output_offset is set to 0, and packet writes begin filling the new output region specified by proc_trace_table_offset.

As packets are written out, each store derives its physical address as follows:

```
trace_store_phys_addr ← Base address from current ToPA table entry +
proc_trace_output_offset
```

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Section 35.2.6.2 for details on STOP.

When an END entry is reached, the processor loads proc_trace_table_base with the base address held in this END entry, thereby moving the current table pointer to this new table. The proc_trace_table_offset is reset to 0, as is the proc_trace_output_offset, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (proc_trace_table_offset = 01FFFFFH). In this case, the proc_trace_table_offset and proc_trace_output_offset are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs

while Intel PT is enabled may return stale values. Like all IA32_RTIT_* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32_RTIT_CTL.TraceEn. This ensures that he output MSR values account for all packets generated to that point, after which the output MSR values will be frozen until tracing resumes. [1]

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 35-2. The size of the address field is determined by the processor's physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

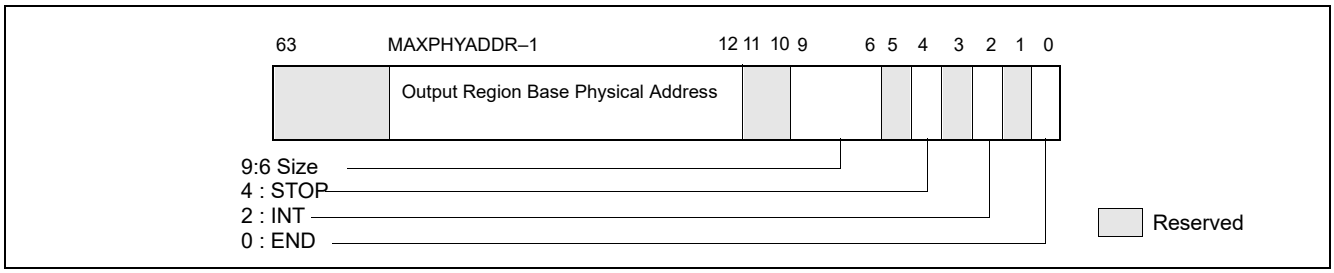


Figure 35-2. Layout of ToPA Table Entry

Table 35-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 35-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

Table 35-3. ToPA Table Entry Fields (Contd.)

ToPA Entry Field	Description
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.

ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32_RTIT_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 35.2.7.4 for details on IA32_RTIT_STATUS.Stopped. This sequence is known as "ToPA Stop".

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32_RTIT_OUTPUT_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset will hold the index value for that entry, and the IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset should be set to the size of the region.

Note that this means the offset pointer is pointing to the next byte after the end of the region, a configuration that would produce an operational error if the configuration remained when tracing is re-enabled with IA32_RTIT_STATUS.Stopped cleared.

ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 834H in x2APIC mode). See *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.
2. Set up an interrupt handler to service the interrupt vector that a ToPA PMI can raise.
3. Set the interrupt flag by executing STI.
4. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32_RTIT_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR (MSR 38EH). Once the ToPA PMI handler has serviced the relevant buffer, writing 1 to bit 55 of the MSR at 390H (IA32_GLOBAL_STATUS_RESET) clears IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI.

Intel PT is not frozen on PMI, and thus the interrupt handler will be traced (though filtering can prevent this). The Freeze_Perfmon_on_PMI and Freeze_LBRs_on_PMI settings in IA32_DEBUGCTL will be applied on ToPA PMI just as on other PMIs, and hence Perfmon counters are frozen.

Assuming the PMI handler wishes to read any buffered packets for persistent output, or wishes to modify any Intel PT MSRs, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

In rare cases, it may be possible to trigger a second ToPA PMI before the first is handled. This can happen if another ToPA region with INT=1 is filled before, or shortly after, the first PMI is taken, perhaps due to EFLAGS.IF being cleared for an extended period of time. This can manifest in two ways: either the second PMI is triggered before the first is taken, and hence only one PMI is taken, or the second is triggered after the first is taken, and thus will be taken when the handler for the first completes. Software can minimize the likelihood of the second case by clearing TraceEn at the beginning of the PMI handler. Further, it can detect such cases by then checking the Interrupt Request Register (IRR) for PMI pending, and checking the ToPA table base and off-set pointers (in IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) to see if multiple entries with INT=1 have been filled.

ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible that, in rare cases, the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 35-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32_RTIT_STATUS.Stopped indication before tracing can resume.

ToPA PMI and XSAVES/XRSTORS State Handling

In some cases the ToPA PMI may be taken after completion of an XSAVES instruction that switches Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, it is recommended that the Intel PT output configuration be modified by altering the ToPA tables themselves, rather than the Intel PT output MSRs.

Table 35-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

Table 35-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS

Pseudo Code Flow

```
IF (IA32_PERF_GLOBAL_STATUS.ToPA)
    Save IA32_RTIT_CTL value;
    IF ( IA32_RTIT_CTL.TraceEN )
        Disable Intel PT by clearing TraceEn;
    FI;
    IF ( there is space available to grow the current ToPA table )
        Add one or more ToPA entries after the last entry in the ToPA table;
        Point new ToPA entry address field(s) to new output region base(s);
    ELSE
        Modify an upcoming ToPA entry in the current table to have END=1;
        IF (output should transition to a new ToPA table )
            Point the address of the "END=1" entry of the current table to the new table base;
        ELSE
            /* Continue to use the current ToPA table, make a circular. */
            Point the address of the "END=1"l entry to the base of the current table;
            Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;
            /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */
        FI;
      FI;
      Restore saved IA32_RTIT_CTL.value;
FI;
```

ToPA Errors

When a malformed ToPA entry is found, an **operation error** results (see Section 35.3.9). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation**.
   This describes cases where a bit marked as reserved in Section 35.2.6.2 above is set to 1.
2. **ToPA alignment violation**.
   This includes cases where illegal ToPA entry base address bits are set to 1:
   a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32_RTIT_OUTPUT_BASE, or from a ToPA entry with END=1.
   b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
   c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset** (if IA32_RTIT_STATUS.Stopped=0).
   IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.
4. **ToPA rules violations**.
   These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
   a. Setting the STOP or INT bit on an entry with END=1.
   b. Setting the END bit in entry 0 of a ToPA table.
   c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
      i) ToPA table entry 1 with END=0.
      ii) ToPA table entry 1 with base address not matching the table base.

In all cases, the error will be logged by setting IA32_RTIT_STATUS.Error, thereby disabling tracing when the problematic ToPA entry is reached (when proc_trace_table_offset points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 35.2.6.4 for details.

A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 35.5.

35.2.6.3 Trace Transport Subsystem

When IA32_RTIT_CTL.FabricEn is set, the IA32_RTIT_CTL.ToPA bit is ignored, and trace output is written to the trace transport subsystem. The endpoints of this transport are platform-specific, and details of configuration options should refer to the specific platform documentation. The FabricEn bit is available to be set if CPUID(EAX=14H,ECX=0):EBX[bit 3] = 1.

35.2.6.4 Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 35-5 summarizes several scenarios.

Table 35-5. Behavior on Restricted Memory Access

Scenario	Description
ToPA output region overlaps with SMRR	Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 35.3.9) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).
ToPA table overlaps with SMRR	The processor will signal an error (Section 35.3.9) and disable tracing when the ToPA read pointer (IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset « 3)) enters the restricted region.

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the IA32_APIC_BASE MSR. For details about the APIC, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. No error is signaled for this case.

Modifications to Restricted Memory Regions

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing IA32_RTIT_CTL.TraceEn.

35.2.7 Enabling and Configuration MSRs

35.2.7.1 General Considerations

Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 35.3.1), RDMSR or WRMSR of the IA32_RTIT_* MSRs will cause #GP.
- A WRMSR to any of these configuration MSRs that begins and ends with IA32_RTIT_CTL.TraceEn set will #GP fault. Packet generation must be disabled before the configuration MSRs can be changed.

Note: Software may write the same value back to IA32_RTIT_CTL without #GP, even if TraceEn=1.

- All configuration MSRs for Intel PT are duplicated per logical processor
- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.
- All configuration MSRs for Intel PT are cleared on a cold RESET.
  - If CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 1, only the TraceEn bit is cleared on warm RESET; though this may have the impact of clearing other bits in IA32_RTIT_STATUS. Other MSR values of the trace configuration MSRs are preserved on warm RESET.
- The semantics of MSR writes to trace configuration MSRs in this chapter generally apply to explicit WRMSR to these registers, using VM-exit or VM-entry MSR load list to these MSRs, XRSTORS with requested feature bit map including XSAVE map component of state_8 (corresponding to IA32_XSS[bit 8]), and the write to IA32_RTIT_CTL.TraceEn by XSAVES (Section 35.3.5.2).

35.2.7.2 IA32_RTIT_CTL MSR

IA32_RTIT_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 35-6.

Table 35-6. IA32_RTIT_CTL MSR

Position	Bit Name	At Reset	Bit Description
0	TraceEn	0	If 1, enables tracing; else tracing is disabled. When this bit transitions from 1 to 0, all buffered packets are flushed out of internal buffers. A further store, fence, or architecturally serializing instruction may be required to ensure that packet data can be observed at the trace endpoint. See Section 35.2.7.3 for details of enabling and disabling packet generation. Note that the processor will clear this bit on #SMI (Section) and warm reset. Other MSR bits of IA32_RTIT_CTL (and other trace configuration MSRs) are not impacted by these events.
1	CYCEn	0	0: Disables CYC Packet (see Section 35.4.2.14). 1: Enables CYC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
2	OS	0	0: Packet generation is disabled when CPL = 0. 1: Packet generation may be enabled when CPL = 0.
3	User	0	0: Packet generation is disabled when CPL > 0. 1: Packet generation may be enabled when CPL > 0.
4	PwrEvtEn	0	0: Power Event Trace packets are disabled. 1: Power Event Trace packets are enabled (see Section 35.2.3, "Power Event Tracing").
5	FUPonPTW	0	0: PTW packets are not followed by FUPs. 1: PTW packets are followed by FUPs.
6	FabricEn	0	0: Trace output is directed to the memory subsystem, mechanism depends on IA32_RTIT_CTL.ToPA. 1: Trace output is directed to the trace transport subsystem, IA32_RTIT_CTL.ToPA is ignored. This bit is reserved if CPUID.(EAX=14H, ECX=0):ECX[bit 3] = 0.
7	CR3Filter	0	0: Disables CR3 filtering. 1: Enables CR3 filtering.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
8	ToPA	0	0: Single-range output scheme enabled if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 1 and IA32_RTIT_CTL.FabricEn=0. 1: ToPA output scheme enabled (see Section 35.2.6.2) if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 1, and IA32_RTIT_CTL.FabricEn=0. Note: WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit and FabricEn would cause #GP, if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 0. WRMSR to IA32_RTIT_CTL that sets this bit causes #GP, if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 0.
9	MTCEn	0	0: Disables MTC Packet (see Section 35.4.2.16). 1: Enables MTC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
10	TSCEn	0	0: Disable TSC packets. 1: Enable TSC packets (see Section 35.4.2.11).
11	DisRETC	0	0: Enable RET compression. 1: Disable RET compression (see Section 35.2.1.2).
12	PTWEn	0	0: PTWRITE packet generation disabled. 1: PTWRITE packet generation enabled (see Table 35-40 "PTW Packet Definition").
13	BranchEn	0	0: Disable COFI-based packets. 1: Enable COFI-based packets: FUP, TIP, TIP.PGE, TIP.PGD, TNT, MODE.Exec, MODE.TSX. see Section 35.2.6 for details on BranchEn.
17:14	MTCFreq	0	Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined: 0: ART(0), 1: ART(1), 2: ART(2), 3: ART(3), 4: ART(4), 5: ART(5), 6: ART(6), 7: ART(7), 8: ART(8), 9: ART(9), 10: ART(10), 11: ART(11), 12: ART(12), 13: ART(13), 14: ART(14), 15: ART(15) Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
18	Reserved	0	Must be 0.
22:19	CycThresh	0	CYC packet threshold, see Section 35.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as $2^{(CycThresh-1)}$. The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384 Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
23	Reserved	0	Must be 0.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
27:24	PSBFreq	0	Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
31:28	Reserved	0	Must be 0.
35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] >= 0.
39:36	ADDR1_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR1_A/B based on the following encodings: 0: ADDR1 range unused. 1: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 2.
43:40	ADDR2_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 3.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
47:44	ADDR3_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 4.
59:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
63:60	Reserved	0	Must be 0.

35.2.7.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 35.4.2.17) will be generated if IA32_RTIT_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 35.4.2).

In addition to the packets discussed above, if and when PacketEn (Section 35.2.5.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a TIP.PGE packet (Section 35.4.2.3) will be generated.

When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the configuration of restricted memory regions). See Section 35.7 for more details of packets that may be generated with modifications to TraceEn.

Disabling Packet Generation

Clearing TraceEn causes any packet data buffered within the logical processor to be flushed out, after which the output MSRs (IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) will have stable values. When output is directed to memory, a store, fence, or architecturally serializing instruction may be required to ensure that the packet data is globally observed. No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

Other Writes to IA32_RTIT_CTL

Any attempt to modify IA32_RTIT_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32_RTIT_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

35.2.7.4 IA32_RTIT_STATUS MSR

The IA32_RTIT_STATUS MSR is readable and writable by software, but some bits (ContextEn, TriggerEn) are read-only and cannot be directly modified. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

Table 35-7. IA32_RTIT_STATUS MSR

Position	Bit Name	At Reset	Bit Description
0	FilterEn	0	This bit is written by the processor, and indicates that tracing is allowed for the current IP, see Section 35.2.5.5. Writes are ignored.
1	ContextEn	0	The processor sets this bit to indicate that tracing is allowed for the current context. See Section 35.2.5.3. Writes are ignored.
2	TriggerEn	0	The processor sets this bit to indicate that tracing is enabled. See Section 35.2.5.2. Writes are ignored.
3	Reserved	0	Must be 0.
4	Error	0	The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA Errors" in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
5	Stopped	0	The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA STOP" in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
31:6	Reserved	0	Must be 0.
48:32	PacketByteCnt	0	This field is written by the processor, and holds a count of packet bytes that have been sent out. The processor also uses this field to determine when the next PSB packet should be inserted. Note that the processor may clear or modify this field at any time while IA32_RTIT_CTL.TraceEn=1. It will have a stable value when IA32_RTIT_CTL.TraceEn=0. See Section 35.4.2.17 for details.
63:49	Reserved	0	Must be 0.

35.2.7.5 IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSRs

The role of the IA32_RTIT_ADDRn_A/B register pairs, for each n, is determined by the corresponding ADDRn_CFG fields in IA32_RTIT_CTL (see Section 35.2.7.2). The number of these register pairs is enumerated by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0].

- Processors that enumerate support for 1 range support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
- Processors that enumerate support for 2 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
- Processors that enumerate support for 3 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B

  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
- Processors that enumerate support for 4 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B

  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B

  IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B

Each register has a single 64-bit field that holds a linear address value. Writes must ensure that the address is in canonical form, otherwise a #GP fault will result.

35.2.7.6 IA32_RTIT_CR3_MATCH MSR

The IA32_RTIT_CR3_MATCH register is compared against CR3 when IA32_RTIT_CTL.CR3Filter is 1. Bits 63:5 hold the CR3 address value to match, bits 4:0 are reserved to 0. For more details on CR3 filtering and the treatment of this register, see Section 35.2.4.2.

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32_RTIT_CR3_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

35.2.7.7 IA32_RTIT_OUTPUT_BASE MSR

This MSR is used to configure the trace output destination, when output is directed to memory (IA32_RTIT_CTL.FabricEn = 0). The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

When the ToPA output scheme is used, the processor may update this MSR when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-8. IA32_RTIT_OUTPUT_BASE MSR

Position	Bit Name	At Reset	Bit Description
6:0	Reserved	0	Must be 0.
MAXPHYADDR-1:7	BasePhysAddr	0	The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA: 0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value. The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 35.2.7.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 35.3.9). 1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See "ToPA Errors" in Section 35.2.6.2 as well as Section 35.3.9.
63:MAXPHYADDR	Reserved	0	Must be 0.

35.2.7.8 IA32_RTIT_OUTPUT_MASK_PTRS MSR

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 35.2.6.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. The size of this field indicates that regions can be of size 128B up to 4GB. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOrTableOffset field, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.

35.2.8 Interaction of Intel® Processor Trace and Other Processor Features

35.2.8.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 35-10 for details.

Table 35-10. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP)
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: ▪ Nested XBEGIN or XACQUIRE lock ▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set) ▪ Non-outermost XEND or XRELEASE lock	None. No change to TSX mode bits for these cases.

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

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 35.5.

35.2.8.5 Intel Software Guard Extensions (SGX)

SGX provides an application with ability to instantiate a protective container (an enclave) with confidentiality and integrity (see *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. Enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.

During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 35.7.

Debug Enclaves

SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by enclave entry or exit. Hence the generation of ContextEn-dependent packets within a debug enclave is allowed.

35.2.8.6 SENTER/ENTERACCS and ACM

GETSEC[SENTER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

35.2.8.7 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 MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 35.2.6.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see "ToPA Errors" in Section 35.2.6.2).
	2	Single-Range Output Supported	1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	3	Output to Trace Transport Subsystem Supported	1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will #GP fault.
	30:4	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.

Table 35-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=1)		Name	Description Behavior
Register	Bits		
EAX	2:0	Number of Address Ranges	A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. **NOTE:** Currently, no processors support more than 4 address ranges.
	15:3	Reserved	
	31:16	Bitmap of supported MTC Period Encodings	The non-zero bit positions indicate the map of supported encoding values for the IA32_RTIT_CTL.MTCFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 1 (MTC Packet generation is supported), otherwise the MTCFreq field is reserved to 0. Each bit position in this field represents 1 encoding value in the 4-bit MTCFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: MTCFreq can be assigned the associated encoding value. 0: MTCFreq cannot be assigned to the associated encoding value. A write to IA32_RTIT_CTLMTCFreq with unsupported encoding will cause #GP fault.
EBX	15:0	Bitmap of supported Cycle Threshold values	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.CycThresh field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Cycle-Accurate Mode is Supported), otherwise the CycThresh field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit CycThresh field (ie, bit 0 is associated with encoding value 0). For each bit: 1: CycThresh can be assigned the associated encoding value. 0: CycThresh cannot be assigned to the associated encoding value. A write to CycThresh with unsupported encoding will cause #GP fault.
	31:16	Bitmap of supported Configurable PSB Frequency encoding	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.PSBFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Configurable PSB is supported), otherwise the PSBFreq field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit PSBFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: PSBFreq can be assigned the associated encoding value. 0: PSBFreq cannot be assigned to the associated encoding value. A write to PSBFreq with unsupported encoding will cause #GP fault.
ECX	31:0	Reserved	
EDX	31:0	Reserved	

35.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an instruction pointer (IP) payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address (Note that in real mode, the CS base address is the value of CS<<4, while in protected mode the CS base address is the base linear address of the segment indicated by the CS register.). Which IP type is in use is indicated by enumeration (see CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31] in Table 35-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases where the CS base address is not 0 and the resolved LIP will not be evident in a trace generated on a CPU that enumerates use of RIP. This is likely to cause problems when attempting to link the trace with the associated binaries.

Note that IP comparison logic, for IP filtering and TraceStop range calculation, is based on the same IP type as these IP packets. For processors that output RIP, the IP comparison mechanism is also based on RIP, and hence on those processors RIP values should be written to IA32_RTIT_ADDRn_[AB] MSRs. This can produce differing behavior if the same trace configuration setting is run on processors reporting different IP types, i.e. CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31]. Care should be taken to check CPUID when configuring IP filters.

35.3.1.2 Model Specific Capability Restrictions

Some processor generations impose restrictions that prevent use of LBRs/BTS/BTM/LERs when software has enabled tracing with Intel Processor Trace. On these processors, when TraceEn is set, updates of LBR, BTS, BTM, LERs are suspended but the states of the corresponding IA32_DEBUGCTL control fields remained unchanged as if it were still enabled. When TraceEn is cleared, the LBR array is reset, and LBR/BTS/BTM/LERs updates will resume. Further, reads of these registers will return 0, and writes will be dropped.

The list of MSRs whose updates/accesses are restricted follows.

- MSR_LASTBRANCH_x_TO_IP, MSR_LASTBRANCH_x_FROM_IP, MSR_LBR_INFO_x, MSR_LASTBRANCH_TOS
- MSR_LER_FROM_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 Info Packet (PIP), if ContextEn=1 and IA32_RTIT_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the "conceal VMX 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).

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+.

35.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor's dedicated internal buffers are all full, an "internal buffer overflow" occurs. On such an overflow packet generation ceases (as packets would need to enter the processor's internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 35.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32_RTIT_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. 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 35.4.2 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of "compound packet events", such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be "coupled" with the TIP packet.

Other packets could be in between the coupled FUP and TIP packet. Timing packets, such as TSC, MTC, CYC, or CBR, could arrive at any time, and hence could intercede in a compound packet event. If an operation changes CR3 or the processor's mode of execution, a state update packet (i.e., PIP or MODE) is generated. The state changes indicated by these intermediate packets should be applied at the IP of the TIP* packet. A summary of compound packet events is provided in Table 35-15; see Section 35.4.2 for more per-packet details and Section 35.7 for more detailed packet generation examples.

Table 35-15. Compound Packet Event Summary

Event Type	Beginning	Middle	End	Comment
Unconditional, uncompressed control-flow transfer	FUP or none	Any combination of PIP, VMCS, MODE.Exec, or none	TIP or TIP.PGD	FUP only for asynchronous events. Order of middle packets may vary. PIP/VMCS/MODE only if the operation modifies the state tracked by these respective packets.
TSX Update	MODE.TSX, and (FUP or none)	None	TIP, TIP.PGD, or none	FUP TIP/TIP.PGD only for TSX abort cases.
Overflow	OVF	PSB, PSBEND, or none	FUP or TIP.PGE	FUP if overflow resolves while ContextEn=1, else TIP.PGE.

35.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 35-3 explains how to interpret them. Packet bits listed as "RSVD" are not guaranteed to be 0.

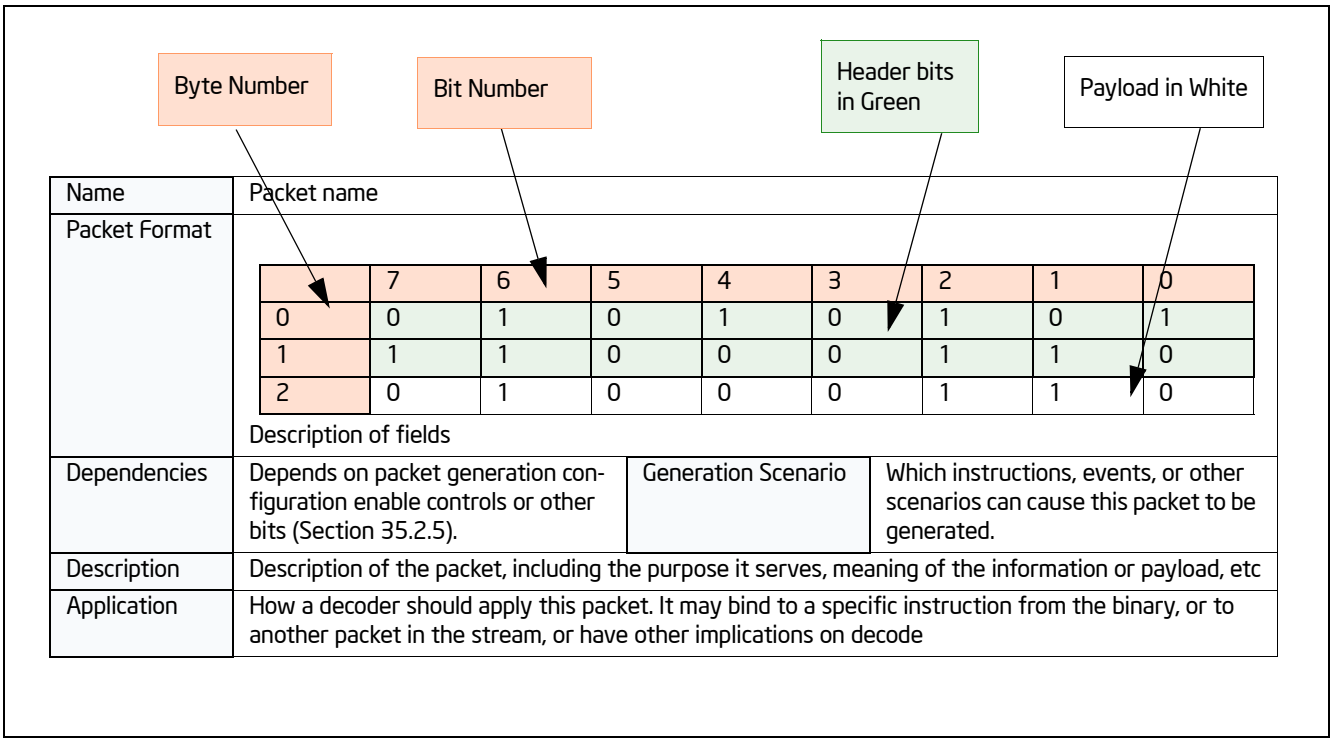


Figure 35-3. Interpreting Tabular Definition of Packet Format

35.4.2.1 Taken/Not-taken (TNT) Packet

Table 35-16. TNT Packet Definition

Name	Taken/Not-taken (TNT) Packet								
Packet Format									
	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

Table 35-16. TNT Packet Definition (Contd.)

<table>
<tr><td></td><td colspan="5">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.</td></tr>
<tr><td></td><td colspan="5">
<table>
<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><th></th></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>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><td></td></tr>
<tr><td>2</td><td>B_{40}</td><td>B_{41}</td><td>B_{42}</td><td>B_{43}</td><td>B_{44}</td><td>B_{45}</td><td>B_{46}</td><td>B_{47}</td><td></td></tr>
<tr><td>3</td><td>B_{32}</td><td>B_{33}</td><td>B_{34}</td><td>B_{35}</td><td>B_{36}</td><td>B_{37}</td><td>B_{38}</td><td>B_{39}</td><td></td></tr>
<tr><td>4</td><td>B_{24}</td><td>B_{25}</td><td>B_{26}</td><td>B_{27}</td><td>B_{28}</td><td>B_{29}</td><td>B_{30}</td><td>B_{31}</td><td></td></tr>
<tr><td>5</td><td>B_{16}</td><td>B_{17}</td><td>B_{18}</td><td>B_{19}</td><td>B_{20}</td><td>B_{21}</td><td>B_{22}</td><td>B_{23}</td><td></td></tr>
<tr><td>6</td><td>B_{8}</td><td>B_{9}</td><td>B_{10}</td><td>B_{11}</td><td>B_{12}</td><td>B_{13}</td><td>B_{14}</td><td>B_{15}</td><td></td></tr>
<tr><td>7</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>B_{5}</td><td>B_{6}</td><td>B_{7}</td><td></td></tr>
</table>
</td></tr>
<tr><td></td><td colspan="5">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.
<table>
<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><th></th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>0</td><td>Short TNT</td></tr>
</table>
</td></tr>
<tr><td></td><td colspan="5">
<table>
<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><th></th></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>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><td></td></tr>
<tr><td>2</td><td>B_{24}</td><td>B_{25}</td><td>B_{26}</td><td>B_{27}</td><td>B_{28}</td><td>B_{29}</td><td>B_{30}</td><td>B_{31}</td><td></td></tr>
<tr><td>3</td><td>B_{16}</td><td>B_{17}</td><td>B_{18}</td><td>B_{19}</td><td>B_{20}</td><td>B_{21}</td><td>B_{22}</td><td>B_{23}</td><td></td></tr>
<tr><td>4</td><td>B_{8}</td><td>B_{9}</td><td>B_{10}</td><td>B_{11}</td><td>B_{12}</td><td>B_{13}</td><td>B_{14}</td><td>B_{15}</td><td></td></tr>
<tr><td>5</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>B_{5}</td><td>B_{6}</td><td>B_{7}</td><td></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><td></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><td></td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="3">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.</td></tr>
<tr><td>Description</td><td colspan="5">Provides the taken/not-taken results for the last 1–N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 35.4.2.2). The TNT payload bits should be interpreted as follows:
▪ 1 indicates a taken conditional branch, or a compressed RET
▪ 0 indicates a not-taken conditional branch</td></tr>
<tr><td>Application</td><td colspan="5">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</td></tr>
</table>

35.4.2.2 Target IP (TIP) Packet

Table 35-17. IP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP (TIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td>Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX[1].</td></tr>
<tr><td>Description</td><td colspan="3">Provides the target for some control flow transfers</td></tr>
<tr><td>Application</td><td colspan="3">Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</td></tr>
<tr><td></td><td colspan="3">The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</td></tr>
</table>

NOTES:

1. 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 35-18. FUP/TIP IP Reconstruction

IPBytes	Uncompressed IP Value							
	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
000b	None, IP is out of context							
001b	Last IP[63:16]						IP Payload[15:0]	
010b	Last IP[63:32]				IP Payload[31:0]			
011b	IP Payload[47] extended		IP Payload[47:0]					
100b	Last IP [63:48]		IP Payload[47:0]					
101b	Reserved							
110b	IP Payload[63:0]							
111b	Reserved							

The processor-internal Last IP state is guaranteed to be reset to zero when a PSB is sent out. This means that the IP that follows the PSB with either be un-compressed (011b or 110b, see Table 35-18), or compressed against zero.

At times, "IPbytes" will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code. In that case, no TargetIP will be included in the packet, since that would expose an instruction point at CPL = 0. When the IP payload is suppressed in this manner, Last IP is not cleared, and instead refers to the last IP packet with a non-zero IPBytes field.

On processors that support a maximum linear address size of 32 bits, IP payloads may never exceed 32 bits (IPBytes <= 010b).

Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don't have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the target of the RET.

In cases where the RET is compressed, the target is guaranteed to match the value produced in 2) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from 2).

The hardware ensure that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because it sends no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32_RTIT_CTL.DisRETC). For processors that employ deferred TIPs (Section 35.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make

clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL's Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior

35.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

Table 35-19. TNT Examples with Deferred TIPs

Code Flow	Packets, Non-Deferred TIPS	Packets, Deferred TIPS
0x1000 cmp %rcx, 0 0x1004 jnz Foo // not-taken 0x1008 jmp %rdx	TNT(0b0), TIP(0x1308)	
0x1308 cmp %rcx, 1 0x130c jnz Bar // not-taken 0x1310 cmp %rcx, 2 0x1314 jnz Baz // taken 0x1500 cmp %eax, 7 0x1504 jg Exit // not-taken 0x1508 jmp %r15	TNT(0b010), TIP(0x1100)	
0x1100 cmp %rbx, 1 0x1104 jg Start // not-taken 0x1108 add %rcx, %eax 0x110c ... // **an asynchronous Interrupt arrives** INThandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)

35.4.2.4 Packet Generation Enable (TIP.PGE)

Table 35-20. TIP.PGE Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP - Packet Generation Enable (TIP.PGE)</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 1</td><td>Generation Scenario</td><td colspan="2">Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins.
This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:
▪ TriggerEn: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR.
▪ FilterEn: This is set when software jumps into the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will be the target of the branch.
▪ ContextEn: This is set on a CPL change, a CR3 write or any other means of changing ContextEn. The IP payload will be the Next IP of the instruction that changes context if it is not a branch, otherwise it will be the target of the branch.</td></tr>
<tr><td>Application</td><td colspan="4">TIP.PGE packets bind to the instruction at the IP given in the payload.</td></tr>
</table>

35.4.2.5 Packet Generation Disable (TIP.PGD)

Table 35-21. TIP.PGD Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP - Packet Generation Disable (TIP.PGD)</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 0</td><td>Generation Scenario</td><td colspan="2">Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn= 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn.
PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:
▪ TriggerEn: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or when IA32_RTIT_STATUS.Stopped is set, or on operational error. The IP payload will be suppressed in this case, and the “IPBytes” field will have the value 0.
▪ FilterEn: This is set when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes = 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.
▪ ContextEn: This can happen on a CPL change, a CR3 write or any other means of changing ContextEn. See Section 35.2.4.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The “IPBytes” field will have value 0.
Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNI bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (ie, SYSCALL when IA32_RTIT_CTL.OS=0, In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.</td></tr>
<tr><td>Application</td><td colspan="4">TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce.
In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.
If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.</td></tr>
</table>

35.4.2.6 Flow Update (FUP) Packet

Table 35-22. FUP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Flow Update (FUP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">IP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">IP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">IP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">IP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">IP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">IP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">IP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">IP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 35.2.4 for details.)</td><td>Generation Scenario</td><td colspan="2">Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit, #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, EENTER, EEXIT, ERESUME, EEE, AEX,[1], INT 0, INT 3, INT n, a WRMSR that disables packet generation.</td></tr>
<tr><td>Description</td><td colspan="4">Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.</td></tr>
<tr><td>Application</td><td colspan="4">FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.
In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 35.4.2.8 for details.
Otherwise, FUPs are part of compound packet events (see Section 35.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.</td></tr>
</table>

NOTES:

1. 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 35-23 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

Table 35-23. FUP Cases and IP Payload

Event	Flow Update IP	Comment
External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), INIT/SIPI	Address of next instruction (Next IP) that would have been executed	Functionally, this matches the LBR FROM field value and also the EIP value which is saved onto the stack.
Exceptions/Faults, Machine check (fault-like)	Address of the instruction which took the exception/fault (Current IP)	This matches the similar functionality of LBR FROM field value and also the EIP value which is saved onto the stack.
Software Interrupt	Address of the software interrupt instruction (Current IP)	This matches the similar functionality of LBR FROM field value, but does not match the EIP value which is saved onto the stack (Next Linear Instruction Pointer - NLIP).
EENTER, EEXIT, ERESUME, Enclave Exiting Event (EEE), AEX[1]	Current IP of the instruction	This matches the LBR FROM field value and also the EIP value which is saved onto the stack.
XACQUIRE	Address of the X* instruction	
XRELEASE, XBEGIN, XEND, XABORT, other transactional abort	Current IP	
#SMI	IP that is saved into SMRAM	
WRMSR that clears TraceEn	Current IP	

NOTES:

1. Information on EENTER, EEXIT, ERESUME, EEE, Asynchronous Enclave eXit (AEX) can be found in *Intel® Software Guard Extensions Programming Reference*.

On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

If there are post-commit task switch faults, the IP value of the FUP will be the original IP when the task switch started. This is the same value as would be seen in the LBR_FROM field. But it is a different value as is saved on the stack or VMCS.

35.4.2.7 Paging Information (PIP) Packet

Table 35-24. PIP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Paging Information (PIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="7">CR3[11:5] or 0</td><td>RSVD/NR</td></tr>
<tr><td>3</td><td colspan="8">CR3[19:12]</td></tr>
<tr><td>4</td><td colspan="8">CR3[27:20]</td></tr>
<tr><td>5</td><td colspan="8">CR3[35:28]</td></tr>
<tr><td>6</td><td colspan="8">CR3[43:36]</td></tr>
<tr><td>7</td><td colspan="8">CR3[51:44]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn && IA32_RTIT_CTL.OS</td><td>Generation Scenario</td><td colspan="2">MOV CR3, Task switch, INIT, SIPI, PSB+, VM exit, VM entry</td></tr>
<tr><td>Description</td><td colspan="4">The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other page modes (32-bit and 4-level paging[1]), these bits are 0.
This packet holds the CR3 address value. It will be generated on operations that modify CR3:
▪ MOV CR3 operation
▪ Task Switch
▪ INIT and SIPI
▪ VM exit, 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
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.
The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.
The PIP packet contains the new CR3 value when CR3 is written.
PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX from PT” VM-execution control is 0 (see Section 35.5.1). All other PIPs clear the NR bit.</td></tr>
<tr><td>Application</td><td colspan="4">The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:
1) If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new CR3/NR values to the TIP payload IP.
2) Otherwise, look for the next MOV CR3, far branch, or VMRESUME/VMLAUNCH in the disassembly, and apply the new CR3 to the next (or target) IP.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

NOTES:

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

35.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

Table 35-25. General Form of MODE Packets

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	Leaf ID			Mode				

The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

MODE.Exec Packet

Table 35-26. MODE.Exec Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.Exec Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>CS.D</td><td>(CS.L & LMA)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="2">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.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L & IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.

<table>
<tr><td>CS.D</td><td>(CS.L & IA32_EFER.LMA)</td><td>Addressing Mode</td></tr>
<tr><td>1</td><td>1</td><td>N/A</td></tr>
<tr><td>0</td><td>1</td><td>64-bit mode</td></tr>
<tr><td>1</td><td>0</td><td>32-bit mode</td></tr>
<tr><td>0</td><td>0</td><td>16-bit mode</td></tr>
</table>

MODE.Exec is sent at the time of a mode change, if PacketEn=1 at the time, or when tracing resumes, if necessary. In the former case, the MODE.Exec packet is generated along with other packets that result from the far transfer operation that changes the mode. In cases where the mode changes while PacketEn=0, the processor will send out a MODE.Exec along with the TIP.PGE when tracing resumes. The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that from the last MODE.Exec packet, if there was no PSB in between.</td></tr>
<tr><td>Application</td><td colspan="4">MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.</td></tr>
</table>

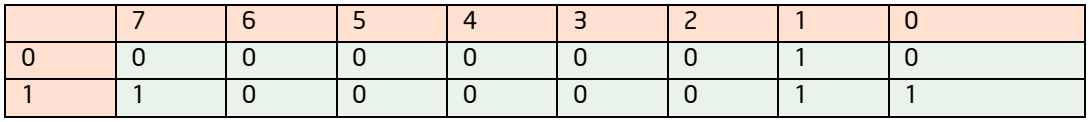
MODE.TSX Packet

Table 35-27. MODE.TSX Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.TSX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>TXAbort</td><td>InTX</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn and ContextEn</td><td>Generation Scenario</td><td colspan="2">XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if INTX changes, Asynchronous TSX Abort, PSB+</td></tr>
<tr><td>Description</td><td colspan="4">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>
<tr><td>TXAbort</td><td>InTX</td><td>Implication</td></tr>
<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>
</table>
</td></tr>
<tr><td>Application</td><td colspan="4">If PacketEn=1, MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP.
MODE.TSX packets may be generated when PacketEn=0, due to FilterEn=0. In this case, only the last MODE.TSX generated before TIP.PGE need be applied.</td></tr>
</table>

35.4.2.9 TraceStop Packet

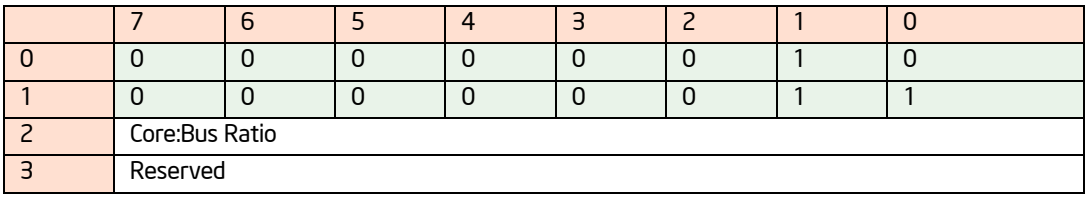
Table 35-28. TraceStop Packet Definition

Name	TraceStop Packet		
Packet Format			
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	Indicates when software has entered a user-configured TraceStop region. When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated. The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 35.2.4.3 for more details.		
Application	If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any). If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.		

	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

35.4.2.10 Core:Bus Ratio (CBR) Packet

Table 35-29. CBR Packet Definition

Name	Core:Bus Ratio (CBR) Packet		
Packet Format			
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	All packets following the CBR represent instructions that executed with the new core:bus ratio, while all preceding packets (aside from timing packets) represent instructions that executed with the prior ratio. There is not a precise IP provided, to which to bind the CBR packet.		

	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							

35.4.2.11 Timestamp Counter (TSC) Packet

Table 35-30. TSC Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Timestamp Counter (TSC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">SW TSC[7:0]</td></tr>
<tr><td>2</td><td colspan="8">SW TSC[15:8]</td></tr>
<tr><td>3</td><td colspan="8">SW TSC[23:16]</td></tr>
<tr><td>4</td><td colspan="8">SW TSC[31:24]</td></tr>
<tr><td>5</td><td colspan="8">SW TSC[39:32]</td></tr>
<tr><td>6</td><td colspan="8">SW TSC[47:40]</td></tr>
<tr><td>7</td><td colspan="8">SW TSC[55:48]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop, This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.</td></tr>
<tr><td>Description</td><td colspan="4">When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.</td></tr>
<tr><td>Application</td><td colspan="4">TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.</td></tr>
</table>

35.4.2.12 Mini Time Counter (MTC) Packet

Table 35-31. MTC Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Mini time Counter (MTC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">CTC[N+7:N]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && TriggerEn</td><td>Generation Scenario</td><td>Periodic, based on the core crystal clock, or Always Running Timer (ART).</td></tr>
<tr><td>Description</td><td colspan="3">When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to (ART >> N) & 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.
MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.
There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that >256 consecutive MTC packets should ever be dropped.</td></tr>
<tr><td>Application</td><td colspan="3">MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.</td></tr>
</table>

35.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 35-32. TMA Packet Definition

<table>
<tr><td>Name</td><td colspan="4">TSC/MTC Alignment (TMA) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="8">CTC[7:0]</td></tr>
<tr><td>3</td><td colspan="8">CTC[15:8]</td></tr>
<tr><td>4</td><td colspan="7">Reserved</td><td>0</td></tr>
<tr><td>5</td><td colspan="8">FastCounter[7:0]</td></tr>
<tr><td>6</td><td colspan="7">Reserved</td><td>FC[8]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Sent with any TSC packet.</td></tr>
<tr><td>Description</td><td colspan="4">The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 35.8.3.2 for details on how to make this calculation.</td></tr>
<tr><td>Application</td><td colspan="4">TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.</td></tr>
</table>

35.4.2.14 Cycle Count Packet (CYC) Packet

Table 35-33. Cycle Count Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Cycle Count (CYC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<tr><td>0</td><td colspan="5">Cycle Counter[4:0]</td><td>Exp</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td colspan="7">Cycle Counter[11:5]</td><td>Exp</td></tr>
<tr><td>2</td><td colspan="7">Cycle Counter[18:12]</td><td>Exp</td></tr>
<tr><td>...</td><td colspan="8">... (if Exp = 1 in the previous byte)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.CYCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 35.3.6 for CYC-eligible packets.</td></tr>
<tr><td>Description</td><td colspan="4">The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length.
If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on.
CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 35.2.7.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 35.3.6.
When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow.
In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor's internal cycle counter doesn't cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen.
When CycThresh>0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone.
When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time.</td></tr>
<tr><td>Application</td><td colspan="4">CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet.
If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet.</td></tr>
</table>

35.4.2.15 VMCS Packet

Table 35-34. VMCS Packet Definition

<table>
<tr><td>Name</td><td colspan="4">VMCS Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">VMCS 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>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn; Also in VMX operation.</td><td>Generation Scenario</td><td colspan="2">Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 35.5).</td></tr>
<tr><td>Description</td><td colspan="4">The VMCS packet provides a VMCS pointer for a decoder to determine the transition of code contexts:
• On a successful VMPTRLD (i.e., a VMPTRLD that doesn't fault, fail, or VM exit), the VMCS packet contains the 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.
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.
VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.</td></tr>
<tr><td>Application</td><td colspan="4">The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique.
When a VMCS packet is encountered, a decoder should do the following:
▪ If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this VMCS is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new VMCS base pointer value to the TIP payload IP.
▪ Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

35.4.2.16 Overflow (OVF) Packet

Table 35-35. OVF Packet Definition

Name	Overflow (OVF) Packet		
Packet Format			
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.		

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	1	1	1	0	0	1	1

35.4.2.17 Packet Stream Boundary (PSB) Packet

Table 35-36. PSB Packet Definition

Name	Packet Stream Boundary (PSB) Packet
Packet Format	

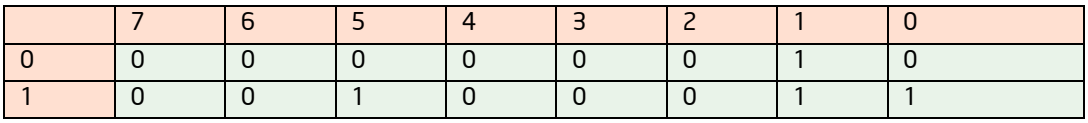
	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0
5	1	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
7	1	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0
9	1	0	0	0	0	0	1	0
10	0	0	0	0	0	0	1	0
11	1	0	0	0	0	0	1	0
12	0	0	0	0	0	0	1	0
13	1	0	0	0	0	0	1	0
14	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	1	0

Table 35-36. PSB Packet Definition (Contd.)

Dependencies	TriggerEn	Generation Scenario	Periodic, based on the number of output bytes generated while tracing. PSB is sent when IA32_RTIT_STATUS.PacketByteCnt=0, and each time it crosses the software selected threshold after that. May be sent for other micro-architectural conditions as well.
Description	PSB is a unique pattern in the packet output log, and hence serves as a sync point for the decoder. It is a pattern that the decoder can search for in order to get aligned on packet boundaries. This packet is periodic, based on the number of output bytes, as indicated by IA32_RTIT_STATUS.PacketByteCnt. The period is chosen by software, via IA32_RTIT_CTL.PSBFreq (see Section 35.2.7.2). Note, however, that the PSB period is not precise, it simply reflects the average number of output bytes that should pass between PSBs. The processor will make a best effort to insert PSB as quickly after the selected threshold is reached as possible. The processor also may send extra PSB packets for some micro-architectural conditions. PSB also serves as the leading packet for a set of "status-only" packets collectively known as PSB+ (Section 35.3.7).		
Application	When a PSB is seen, the decoder should interpret all following packets as "status only", until either a PSBEND or OVF packet is encountered. "Status only" implies that the binding and ordering rules to which these packets normally adhere are ignored, and the state they carry can instead be applied to the IP payload in the FUP packet that is included.		

35.4.2.18 PSBEND Packet

Table 35-37. PSBEND Packet Definition

Name	PSBEND Packet		
Packet Format			
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 35-38. MNT Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Maintenance (MNT) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<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>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Implementation specific.</td></tr>
<tr><td>Description</td><td colspan="4">This packet is generated by hardware, the payload meaning is model-specific.</td></tr>
<tr><td>Application</td><td colspan="4">Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.</td></tr>
</table>

35.4.2.20 PAD Packet

Table 35-39. PAD Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PAD Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<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>
<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>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Implementation specific</td></tr>
<tr><td>Description</td><td colspan="4">PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.</td></tr>
<tr><td>Application</td><td colspan="4">Ignore PAD packets.</td></tr>
</table>

35.4.2.21 PTWRITE (PTW) Packet

Table 35-40. PTW Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PTW Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td colspan="2">PayloadBytes</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">Payload[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Payload[15:8]</td></tr>
<tr><td>4</td><td colspan="8">Payload[23:16]</td></tr>
<tr><td>5</td><td colspan="8">Payload[31:24]</td></tr>
<tr><td>6</td><td colspan="8">Payload[39:32]</td></tr>
<tr><td>7</td><td colspan="8">Payload[47:40]</td></tr>
<tr><td>8</td><td colspan="8">Payload[55:48]</td></tr>
<tr><td>9</td><td colspan="8">Payload[63:56]</td></tr>
</table>
</td></tr>
<tr><td></td><td colspan="4">The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:
<table>
<tr><th>PayloadBytes</th><th>Bytes of Payload</th></tr>
<tr><td>‘00</td><td>4</td></tr>
<tr><td>‘01</td><td>8</td></tr>
<tr><td>‘10</td><td>Reserved</td></tr>
<tr><td>‘11</td><td>Reserved</td></tr>
</table>
IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn & FilterEn & PTWEn</td><td>Generation Scenario</td><td colspan="2">PTWRITE Instruction</td></tr>
<tr><td>Description</td><td colspan="4">Contains the value held in the PTWRITE operand.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="4">Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.</td></tr>
</table>

35.4.2.22 Execution Stop (EXSTOP) Packet

Table 35-41. EXSTOP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">EXSTOP Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
IP bit indicates if a FUP will follow.
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td>C-state entry, P-state change, or other processor clock power-down. Includes :
▪ Entry to C-state deeper than C0.0
▪ TM1/2
▪ STPCLK#
▪ Frequency change due to IA32_CLOCK_MODULATION, Turbo</td></tr>
<tr><td>Description</td><td colspan="3">This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes.
If EXSTOP is generated while ContextEn is set, the IP bit will be set, and EXSTOP will be followed by a FUP packet containing the IP at which execution stopped. More precisely, this will be the IP of the oldest instruction that has not yet completed.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">If a FUP follows EXSTOP (hence IP bit set), the EXSTOP can be bound to the FUP IP. Otherwise the IP is not known. Time of powerdown can be inferred from the preceding CYC, if CYCEn=1. Combined with the TSC at the time of wake (if TSCEn=1), this can be used to determine the duration of the powerdown.</td></tr>
</table>

35.4.2.23 MWAIT Packet

Table 35-42. MWAIT Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MWAIT Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">MWAIT Hints[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Reserved</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="6">Reserved</td><td colspan="2">EXT[1:0]</td></tr>
<tr><td>7</td><td colspan="8">Reserved</td></tr>
<tr><td>8</td><td colspan="8">Reserved</td></tr>
<tr><td>9</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn & Contex-tEn</td><td>Generation Scenario</td><td colspan="2">MWAIT instruction, or I/O redirection to MWAIT, that complete without fault or VMexit.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that an MWAIT operation to C-state deeper than C0.0 completed. The MWAIT hints and extensions passed in by software are exposed in the payload.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="4">The MWAIT packet should bind to the IP of the next FUP, which will be the IP of the instruction that caused the MWAIT. This FUP will be shared with EXSTOP.</td></tr>
</table>

35.4.2.24 Power Entry (PWRE) Packet

Table 35-43. PWRE Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PWRE Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td>HW</td><td colspan="7">Reserved</td></tr>
<tr><td>3</td><td colspan="4">Resolved Thread C-State</td><td colspan="4">Resolved Thread Sub C-State</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td colspan="2">Transition to a C-state deeper than C0.0.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.
Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core.
If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, HLT, or shut-down), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 14.5, “Hardware Duty Cycling (HDC)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B) is an example of such a case.</td></tr>
<tr><td>Application</td><td colspan="4">When transitioning from C0.0 to a deeper C-state, the PWRE packet will be followed by an EXSTOP. If that EXSTOP packet has the IP bit set, then the following FUP will provide the IP at which the C-state entry occurred. Subsequent PWRE packets generated before the next PWRX should bind to the same IP.</td></tr>
</table>

35.4.2.25 Power Exit (PWRX) Packet

Table 35-44. PWRX Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PWRX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="4">Last Core C-State</td><td colspan="4">Deepest Core C-State</td></tr>
<tr><td>3</td><td colspan="4">Reserved</td><td colspan="4">Wake Reason</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td colspan="2">Transition from a C-state deeper than C0.0 to C0.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates processor return to thread C0 from a C-state deeper than C0.0.
The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested.
The Wake Reason field is one-hot, encoded as follows:
<table>
<tr><th>Bit</th><th>Field</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt</td><td>Wake due to external interrupt received.</td></tr>
<tr><td>1</td><td>Reserved</td><td></td></tr>
<tr><td>2</td><td>Store to Monitored Address</td><td>Wake due to store to monitored address.</td></tr>
<tr><td>3</td><td>HW Wake</td><td>Wake due to hardware autonomous condition, such as HDC.</td></tr>
</table>
</td></tr>
<tr><td>Application</td><td colspan="4">PWRX will always apply to the same IP as the PWRE. The time of wake can be discerned from (optional) timing packets that precede PWRX.</td></tr>
</table>

35.5 TRACING IN VMX OPERATION

On processors that IA32_VMX_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. A series of mechanisms exist to allow the VMM to configure tracing based on the desired trace domain, and on the consumer of the trace output. The VMM can configure specific VMX controls to control what virtualization-specific data are included within the trace packets (see Section 35.5.1 for details). 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). These configuration options are summarized in Table 35-45. Table 35-45 covers common Intel PT usages while SMIs are handled by the default SMM treatment. Tracing with SMM Transfer Monitor is described in Section 35.6.

Table 35-45. Common Usages of Intel PT and VMX

Target Domain	Output Consumer	Virtualize Output	Configure VMX Controls	TraceEN Configuration	Save/Restore MSR states of Trace Configuration
System-Wide (VMM + VMs)	Host	N/A	Default setting (no suppression)	WRMSR or XRSTORS by Host	N/A
VMM Only	Intel PT Aware VMM	N/A	Enable suppression	Use VMX MSR-load areas to disable tracing in VM, enable tracing on VM exits	N/A
VM Only	Intel PT Aware VMM	N/A	Enable suppression	Use VMX MSR-load areas to enable tracing in VM, disable tracing on VM exits	N/A
Intel PT Aware Guest(s)	Per Guest	VMM adds trace output virtualization	Enable suppression	Use VMX MSR-load areas to enable tracing in VM, disable tracing on VM exits	VMM updates guest state on VM exits due to XRSTORS

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 35-46. 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 35-46 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 using MSR load areas across VMX transitions.

For tracing scenarios that collect packets only within VMX root operation or only within VMX non-root operation, the VMM can use the MSR load areas to toggle IA32_RTIT_CTL.TraceEn.

35.5.2.1 System-Wide Tracing

When a host or VMM configures Intel PT to collect trace packets of the entire system, it can leave the relevant VMX controls clear to allow VMX-specific packets to provide information across VMX transitions. The VMX MSR-load areas need not be used to load Intel PT MSRs on VM exits or VM entries.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 35-47.

Table 35-47. Packets on VMX Transitions (System-Wide Tracing)

Event	Packets	Description
VM exit	FUP(GuestIP)	The FUP indicates at which point in the guest flow the VM exit occurred. This is important, since VM exit can be an asynchronous event. The IP will match that written into the VMCS.
	PIP(HostCR3, NR=0)	The PIP packet provides the new host CR3 value, as well as indication that the logical processor is entering VMX root operation. This allows the decoder to identify the change of executing context from guest to host and load the appropriate set of binaries to continue decode.
	TIP(HostIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	PIP(GuestCR3, NR=1)	The PIP packet provides the new guest CR3 value, as well as indication that the logical processor is entering VMX non-root operation. This allows the decoder to identify the change of executing context from host to guest and load the appropriate set of binaries to continue decode.
	TIP(GuestIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX non-root operation. This should match the 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 Host-Only Tracing

When trace packets in VMX non-root operation are not desired, the VMM can use the VM-entry MSR-load area to load IA32_RTIT_CTL (clearing TraceEn) to disable trace-packet generation in guests, and use the VM-exit MSR-load area to load IA32_RTIT_CTL to set TraceEn.

When tracing only the host, the decoder does not need information about the guests, and the VMX controls for suppressing VMX-specific packets can be set to reduce the packets generated. VMCS packets will still be generated on execution of VMPTRLD and in PSB+ generated in the host, but these will be unused by the decoder.

The packets of interests to a decoder when trace packets are collected for host-only tracing are shown in Table 35-48.

Table 35-48. Packets on VMX Transitions (Host-Only Tracing)

Event	Packets	Description
VM exit	TIP.PGE(HostIP)	The TIP.PGE indicates that trace packet generation is enabled and gives the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	TIP.PGD()	The TIP indicates that trace packet generation was disabled. This ensure that all buffered packets are flushed out.

35.5.2.3 Guest-Only Tracing

A VMM can configure trace-packet generation while in VMX non-root operation for guests executing normally. This is accomplished by utilizing the VMX MSR-load areas on VM exits and VM entries to limit trace-packet generation to the guest environment.

For this usage, the VM-entry MSR load area is programmed to enable trace packet generation; the VM-exit MSR load area is used 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 35-46.

35.5.2.4 Virtualization of Guest Output Packet Streams

Each Intel PT aware guest OS can produce one or more output packet streams to destination addresses specified as guest physical address using by context-switching IA32_RTIT_OUTPUT_BASE within the guest. The processor generates trace packets to the physical address specified in IA32_RTIT_OUTPUT_BASE, and those specified in the ToPA tables. Thus, a VMM that supports Intel PT aware guest OS may wish to virtualize the output configurations of IA32_RTIT_OUTPUT_BASE and ToPA for each trace configuration state of all the guests.

35.5.2.5 Emulation of Intel PT Traced State

If a VMM emulates an element of processor state by taking a VM exit on reads and/or writes to that piece of state, and the state element impacts Intel PT packet generation or values, it may be incumbent upon the VMM to insert or modify the output trace data.

If a VM exit is taken on a guest write to CR3 (including “MOV CR3” as well as task switches), the PIP packet normally generated on the CR3 write will be missing.

To avoid decoder confusion when the guest trace is decoded, the VMM should emulate the missing PIP by writing it into the guest output buffer. If the guest CR3 value is manipulated, the VMM may also need to manipulate the IA32_RTIT_CR3_MATCH value, in order to ensure the trace behavior matches the guest's expectation.

Similarly, if a VMM emulates the TSC value by taking a VM exit on RDTSC, the TSC packets generated in the trace may mismatch the TSC values returned by the VMM on RDTSC. To ensure that the trace can be properly aligned

with software logs based on RDTSC, the VMM should either make corresponding modifications to the TSC packet values in the guest trace, or use mechanisms such as TSC offsetting or TSC scaling in place of exiting.

35.5.2.6 TSC Scaling

When TSC scaling is enabled for a guest using Intel PT, the VMM should ensure that the value of Maximum Non-Turbo Ratio[15:8] in MSR_PLATFORM_INFO (MSR 0CEH) and the TSC/"core crystal clock" ratio (EBX/EAX) in CPUID leaf 15H are set in a manner consistent with the resulting TSC rate that will be visible to the VM. This will allow the decoder to properly apply TSC packets, MTC packets (based on the core crystal clock or ART, whose frequency is indicated by CPUID leaf 15H), and CBR packets (which indicate the ratio of the processor frequency to the Max Non-Turbo frequency). Absent this, or separate indication of the scaling factor, the decoder will be unable to properly track time in the trace. See Section 35.8.3 for details on tracking time within an Intel PT trace.

35.5.2.7 Failed VM Entry

The packets generated by a failed VM entry depend both on the VMCS configuration, as well as on the type of failure. The results to expect are summarized in the table below. Note that packets in *italics* may or may not be generated, depending on implementation choice, and the point of failure.

Table 35-49. Packets on a Failed VM Entry

Usage Model	Entry Configuration	Early Failure (fall through to next IP)	Late Failure (VM-exit like)
System-Wide	No use of 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	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	VM-entry MSR-load area used to set TraceEn	None	None

35.5.2.8 VMX Abort

VMX abort conditions take the processor into a shutdown state. On a VM exit that leads to VMX abort, some packets (FUP, PIP) may be generated, but any expected TIP, TIP.PGE, or TIP.PGD may be dropped.

35.6 TRACING AND SMM TRANSFER MONITOR (STM)

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 not saved and then cleared. Software can save the state of the trace configuration MSRs and clear TraceEn using the MSR load/save lists.

35.7 PACKET GENERATION SCENARIOS

Table 35-50 and Table 35-52 illustrate the packets generated in various scenarios. In the heading row, PacketEn is abbreviated as PktEn, ContextEn as CntxEn. Note that this assumes that TraceEn=1 in IA32_RTIT_CTL, while TriggerEn=1 and Error=0 in IA32_RTIT_STATUS, unless otherwise specified. Entries that do not matter in packet generation are marked "D.C." Packets followed by a "?" imply that these packets depend on additional factors, which are listed in the "Other Dependencies" column.

In Table 35-50, PktEn is evaluated based on TiggerEn & ContextEn & FilterEn & BranchEn.

Table 35-50. Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
1a	Normal non-jump operation	0	0	D.C.		None
1b	Normal non-jump operation	1	1	1		None
2a	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR
2b	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	PSB, PSBEND (see Section 35.4.2.17)
2d	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	1	1	TSC if TSCEn=1; TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, MODE.Exec, TIP.PGE(NLIP)
2e	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	1	1		MODE.Exec, TIP.PGE(NLIP), PSB, PSBEND (see Section 35.4.2.8, 35.4.2.7, 35.4.2.13,35.4.2.15, 35.4.2.17)
3a	WRMSR that changes TraceEn 1 -> 0	0	0	D.C.		None
3b	WRMSR that changes TraceEn 1 -> 0	1	0	D.C.		FUP(CLIP), TIP.PGD()
5a	MOV to CR3	0	0	0		None
5f	MOV to CR3	0	0	1	TraceStop if executed in a TraceStop region	PIP(NewCR3,NR?), TraceStop?
5b	MOV to CR3	0	1	1	*PIP.NR=1 if not in root operation and 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()
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.PGE(NLIP), TraceStop?
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?)
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)

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
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)
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?
7d	Conditional taken jump or compressed RET that fills up the internal TNT buffer	1	1	1		TNT
8a	Conditional non-taken jump	0	0	D.C.		None
8d	Conditional not-taken jump that fills up the internal TNT buffer	1	1	1		TNT
9a	Near indirect jump (JMP, CALL, or uncompressed RET)	0	0	D.C.		None
9b	Near indirect jump (JMP, CALL, or uncompressed RET)	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
9c	Near indirect jump (JMP, CALL, or uncompressed RET)	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
9d	Near indirect jump (JMP, CALL, or uncompressed RET)	1	1	1		TIP(BLIP)
10a	Far Branch (CALL/JMP/RET)	0	0	0		None
10f	Far Branch (CALL/JMP/RET)	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation and the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
10b	Far Branch (CALL/JMP/RET)	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation and 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)	1	0	0		TIP.PGD()

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
10d	Far Branch (CALL/JMP/RET)	1	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation and 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)	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)
11a	HW Interrupt	0	0	0		None
11f	HW Interrupt	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation and the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
11b	HW Interrupt	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation and 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()
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

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
11e	HW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation and 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)
12a	SW Interrupt	0	0	0		None
12f	SW Interrupt	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation and the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
12b	SW Interrupt	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation and 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?
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)
13a	Exception/Fault	0	0	0		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
13f	Exception/Fault	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation and the "conceal VMX from PT" VM-execution control is 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
13b	Exception/Fault	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation and 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)
14a	SMI (TraceEn cleared)	0	0	D.C.		None
14b	SMI (TraceEn cleared)	1	0	0		FUP(SMRAM,LIP), TIP.PGD()
14f	SMI (TraceEn cleared)	1	0	1		NA
14c	SMI (TraceEn cleared)	1	1	1		NA
15a	RSM, TraceEn restored to 0	0	0	0		None
15b	RSM, TraceEn restored to 1	0	0	D.C.		See WRMSR cases for packets on enable
15c	RSM, TraceEn restored to 1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15e	RSM (TraceEn=1, goes to shutdown)	1	0	0		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
15f	RSM (TraceEn=1, goes to shutdown)	1	0	1		None
15d	RSM (TraceEn=1, goes to shutdown)	1	1	1		None
16i	VM exit	0	0	0		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
16j	VM exit, ContextEN 1->0	1	0	0		FUP(VMCSg.LIP), TIP.PGD
16g	VM exit	1	0	1	*PIP if OF=1 and 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)
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

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
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)
17j	VM entry, ContextEN 0->1	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec, TIP.PGE(VMCSg.LIP)
17g	VM entry, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1 and 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)
20a	EENTER/ERESUME to non-debug enclave	0	0	0		None
20c	EENTER/ERESUME to non-debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
21a	EEXIT from non-debug enclave	0	0	D.C.		None
21b	EEXIT from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
22a	AEX/EEE from non-debug enclave	0	0	D.C.		None
22b	AEX/EEE from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
23a	EENTER/ERESUME to debug enclave	0	0	D.C.		None
23b	EENTER/ERESUME to debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
23c	EENTER/ERESUME to debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
23d	EENTER/ERESUME to debug enclave	0	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?
23e	EENTER/ERESUME to debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
24f	EEXIT from debug enclave	0	0	D.C.		None
24b	EEXIT from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
24d	EEXIT from debug enclave	1	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?
24e	EEXIT from debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
25a	AEX/EEE from debug enclave	0	0	D.C.		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
25b	AEX/EEE from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
25d	AEX/EEE from debug enclave	1	0	1	*For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), TIP.PGD(AEP.LIP)
25e	AEX/EEE from debug enclave	1	1	1	*MODE.Exec if the value is different, since last TIP.PGD *For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), MODE.Exec?, TIP(AEP.LIP)
26a	XBEGIN/XACQUIRE	0	0	D.C.		None
26d	XBEGIN/XACQUIRE that does not set InTX	1	1	1		None
26e	XBEGIN/XACQUIRE that sets InTX	1	1	1		MODE(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0	D.C.		None
27d	XEND/XRELEASE that does not clear InTX	1	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1	1		MODE(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TraceStop?
28b	XABORT(Async XAbort, or other)	0	1	1		MODE(InTX=0, TXAbort=1), TIP.PGE(BLIP)
28c	XABORT(Async XAbort, or other)	1	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	BIP(0), TraceStop?
30c	INIT (BSP)	0	1	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, PIP(0), TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0	0		FUP(NLIP), TIP.PGD()
30e	INIT (BSP)	1	0	1	* PIP if OS=1 *TraceStop if RESET.LIP is in a TraceStop region	FUP(NLIP), PIP(0), TIP.PGD, TraceStop?
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

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
31b	INIT (AP, goes to wait-for-SIPI)	1	D.C.	D.C.	* PIP if OS=1	FUP(NLIP), PIP(0)
32a	SIPI	0	0	0		None
32c	SIPI	0	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(SIPI-LIP)
32d	SIPI	1	0	0		TIP.PGD
32e	SIPI	1	0	1	*TraceStop if SIPI LIP is in a TraceStop region	TIP.PGD(SIPILIP); TraceS-top?
32f	SIPI	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP(SIPILIP)
33a	MWAIT (to C0)	D.C.	D.C.	D.C.		None
33b	MWAIT (to higher-numbered C-State, packet sent on wake)	D.C.	D.C.	D.C.	*TSC if TSCEn=1 *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PwrEvtEn).

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.1	MWAIT or I/O redir to MWAIT, gets #UD or #GP fault	dc	dc	dc		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	dc	dc	dc		See VM exit examples (16[a-z] in Table 35-50) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests C0, or monitor not armed, or VMX virtual-interrupt delivery	dc	dc	dc		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	dc	1		MWAIT(Cy), PWRE(Cx), EXSTOP(IP), FUP(CLIP)
16.5a	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP, TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.5b	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	dc	1		MWAIT(5), PWRE(C1), EXSTOP(IP), FUP(CLIP)

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.9a	HLT, Triple-fault shutdown, #MC with CR4.MCE=0, RSM to Cx (x>0)	dc	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	dc	dc			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	dc	0	0		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.10b	VMX abort	dc	dc	1		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	dc	0	0		See "RSM to Shutdown" (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.11b	RSM to Shutdown	dc	dc	1		See "RSM to Shutdown" (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.12a	INIT (BSP)	dc	0	0		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP
16.12b	INIT (BSP)	dc	dc	1		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.13a	INIT (AP, goes to Wait-for-SIPI)	dc	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.13b	INIT (AP, goes to Wait-for-SIPI)	dc	dc	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP, TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.14b	Hardware Duty Cycling (HDC)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	dc	0	0		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP
16.15b	VM entry to HLT or Shutdown	dc	dc	1		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STPCLK#	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP, TSC?, TMA?, CBR
16.16b	EIST in C0, S1/TM1/TM2, or STPCLK#	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	dc	dc	dc		None
16.18	INTR during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x1) See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.18	SMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See "HW Interrupt" (cases 14[a-z] in Table 35-50) for BranchEn packets that follow.

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.19	NMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.2	Store to monitored address during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x4)
16.22	#MC, IERR, TSC deadline timer expiration, or APIC counter under-flow during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PTWEn).

Table 35-52. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	dc	dc	dc		None
16.24b	PTWRITE rm32/64, no fault	dc	0	0		None
16.24d	PTWRITE rm32, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	dc	dc	dc		See "Exception/fault" (cases 13[a-z] in Table 35-50) for BranchEn packets.

35.8 SOFTWARE CONSIDERATIONS

35.8.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section , SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follows:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM
   - Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
   - For processors on which Intel PT and LBR use are mutually exclusive (see Section 35.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

35.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix "IA32_RTIT_" in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.

- Relinquish ownership of the trace configuration MSRs by clearing the "enabled bits" of those configuration MSRs.

35.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

- Processor timestamp counter

  This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] =0.
- Core crystal clock

  The ratio of the core crystal clock to timestamp counter frequency is known as P, and can 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:

```
TimeStampValue = (CoreCrystalClockValue * P) + AdjustedProcessorCycles + Software_Offset;
```

The CoreCrystalClockValue can provide the coarse-grained component of the TSC value. P, or the TSC/"core crystal clock" ratio, can be derived from CPUID leaf 15H, as described in Section 35.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.

The Software_Offsets component includes software offsets that are factored into the timestamp value, such as IA32_TSC_ADJUST.

35.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 35.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the CoreCrystalClockValue. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

(CTC_B - CTC_A), where CTC_i = MTC_i[15:8] << IA32_RTIT_CTL.MTCFreq and i = A, B.

The time from a TSC packet to the subsequent MTC packet can be calculated using the TMA packet that follows the TSC packet. The TMA packet provides both the crystal clock value (lower 16 bits, in the CTC field) and the AdjustedProcessorCycles value (in the FastCounter field) that can be used in the calculation of the corresponding core crystal clock value of the TSC packet.

When the next MTC after a pair of TSC/TMA is seen, the number of crystal clocks passed since the TSC packet can be calculated by subtracting the TMA.CTC value from the time indicated by the MTC_{Next} packet by

CTC_{Delta}[15:0] = (CTC_{Next}[15:0] - TMA.CTC[15:0]), where CTC_{Next} = $MTC_{Payload}$ << IA32_RTIT_CTL.MTCFreq.

The TMA.FastCounter field provides the fractional component of the TSC packet into the next crystal clock cycle.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the adjusted_processor_cycles component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the core:bus ratio value given in the CBR packet, by multiplying the accumulated cycle count value by $P1/CBR_{payload}$.

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 35.5.2.6 for details.

35.8.3.4 Calculating Frequency with Intel PT

Because Intel PT can provide both wall-clock time and processor clock cycle time, it can be used to measure the processor core clock frequency. Either TSC or MTC packets can be used to track the wall-clock time. By using CYC packets to count the number of processor core cycles that pass in between a pair of wall-clock time packets, the ratio between processor core clock frequency and TSC frequency can be derived. If the P1 frequency is known, it can be applied to determine the CPU frequency. See Section 35.8.3.1 above for details on the relationship between TSC, MTC, and CYC.

23.Updates to Chapter 36, Volume 3D

Change bars show changes to Chapter 36 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Various updates throughout chapter regarding Intel SGX and new Intel SGX VM Over-subscription feature.

CHAPTER 36
INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

36.1 OVERVIEW

Intel® Software Guard Extensions (Intel® SGX) is a set of instructions and mechanisms for memory accesses added to Intel® Architecture processors. Intel SGX can encompass two collections of instruction extensions, referred to as SGX1 and SGX2, see Table 36-1 and Table 36-2. The SGX1 extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave is a protected area in the application's address space (see Figure 36-1), which provides confidentiality and integrity even in the presence of privileged malware. Accesses to the enclave memory area from any software not resident in the enclave are prevented. The SGX2 extensions allow additional flexibility in runtime management of enclave resources and thread execution within an enclave.

Chapter 37 covers main concepts, objects and data structure formats that interact within the Intel SGX architecture. Chapter 38 covers operational aspects ranging from preparing an enclave, transferring control to enclave code, and programming considerations for the enclave code and system software providing support for enclave execution. Chapter 39 describes the behavior of Asynchronous Enclave Exit (AEX) caused by events while executing enclave code. Chapter 40 covers the syntax and operational details of the instruction and associated leaf functions available in Intel SGX. Chapter 41 describes interaction of various aspects of IA32 and Intel® 64 architectures with Intel SGX. Chapter 42 covers Intel SGX support for application debug, profiling and performance monitoring.

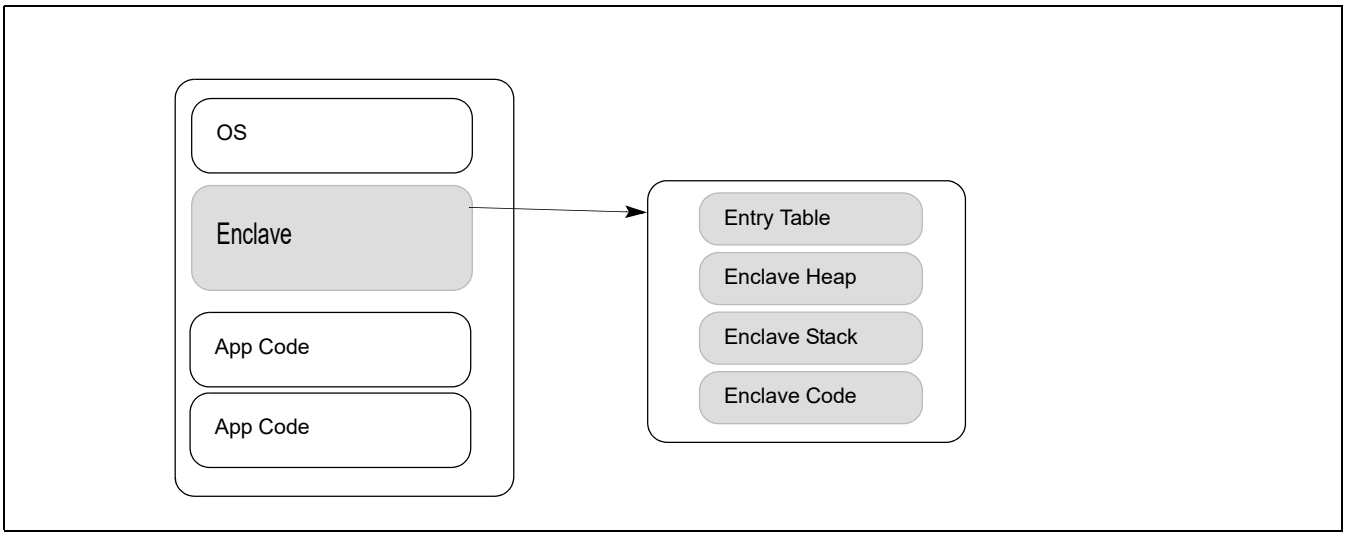


Figure 36-1. An Enclave Within the Application's Virtual Address Space

36.2 ENCLAVE INTERACTION AND PROTECTION

Intel SGX allows the protected portion of an application to be distributed in the clear. Before the enclave is built, the enclave code and data are free for inspection and analysis. The protected portion is loaded into an enclave where its code and data is measured. Once the application's protected portion of the code and data are loaded into an enclave, it is protected against external software access. An enclave can prove its identity to a remote party and provide the necessary building-blocks for secure provisioning of keys and credentials. The application can also request an enclave-specific and platform-specific key that it can use to protect keys and data that it wishes to store outside the enclave.[1]

1. For additional information, see white papers on Intel SGX at http://software.intel.com/en-us/intel-isa-extensions.

Intel SGX introduces two significant capabilities to the Intel Architecture. First is the change in enclave memory access semantics. The second is protection of the address mappings of the application.

36.3 ENCLAVE LIFE CYCLE

Enclave memory management is divided into two parts: address space allocation and memory commitment. Address space allocation is the specification of the range of logical addresses that the enclave may use. This range is called the ELRANGE. No actual resources are committed to this region. Memory commitment is the assignment of actual memory resources (as pages) within the allocated address space. This two-phase technique allows flexibility for enclaves to control their memory usage and to adjust dynamically without overusing memory resources when enclave needs are low. Commitment adds physical pages to the enclave. An operating system may support separate allocate and commit operations.

During enclave creation, code and data for an enclave are loaded from a clear-text source, i.e. from non-enclave memory.

Untrusted application code starts using an initialized enclave typically by using the EENTER leaf function provided by Intel SGX to transfer control to the enclave code residing in the protected Enclave Page Cache (EPC). The enclave code returns to the caller via the EEXIT leaf function. Upon enclave entry, control is transferred by hardware to software inside the enclave. The software inside the enclave switches the stack pointer to one inside the enclave. When returning back from the enclave, the software swaps back the stack pointer then executes the EEXIT leaf function.

On processors that support the SGX2 extensions, an enclave writer may add memory to an enclave using the SGX2 instruction set, after the enclave is built and running. These instructions allow adding additional memory resources to the enclave for use in such areas as the heap. In addition, SGX2 instructions allow the enclave to add new threads to the enclave. The SGX2 features provide additional capabilities to the software model without changing the security properties of the Intel SGX architecture.

Calling an external procedure from an enclave could be done using the EEXIT leaf function. Software would use EEXIT and a software convention between the trusted section and the untrusted section.

An active enclave consumes resources from the Enclave Page Cache (EPC, see Section 36.5). Intel SGX provides the EREMOVE instruction that an EPC manager can use to reclaim EPC pages committed to an enclave. The EPC manager uses EREMOVE on every enclave page when the enclave is torn down. After successful execution of EREMOVE the EPC page is available for allocation to another enclave.

36.4 DATA STRUCTURES AND ENCLAVE OPERATION

There are 2 main data structures associated with operating an enclave, the SGX Enclave Control Structure (SECS, see Section 37.7) and the Thread Control Structure (TCS, see Section 37.8).

There is one SECS for each enclave. The SECS contains meta-data about the enclave which is used by the hardware and cannot be directly accessed by software. Included in the SECS is a field that stores the enclave build measurement value. This field, MRENCLAVE, is initialized by the ECREATE instruction and updated by every EADD and EEXTEND. It is locked by EINIT.

Every enclave contains one or more TCS structures. The TCS contains meta-data used by the hardware to save and restore thread specific information when entering/exiting the enclave. There is one field, FLAGS, that may be accessed by software. This field can only be accessed by debug enclaves. The flag bit, DBGOPTIN, allows to single step into the thread associated with the TCS. (see Section 37.8.1)

The SECS is created when ECREATE (see Table 36-1) is executed. The TCS can be created using the EADD instruction or the SGX2 instructions (see Table 36-2).

36.5 ENCLAVE PAGE CACHE

The Enclave Page Cache (EPC) is the secure storage used to store enclave pages when they are a part of an executing enclave. For an EPC page, hardware performs additional access control checks to restrict access to the page. After the current page access checks and translations are performed, the hardware checks that the EPC page

is accessible to the program currently executing. Generally an EPC page is only accessed by the owner of the executing enclave or an instruction which is setting up an EPC page

The EPC is divided into EPC pages. An EPC page is 4KB in size and always aligned on a 4KB boundary.

Pages in the EPC can either be valid or invalid. Every valid page in the EPC belongs to one enclave instance. Each enclave instance has an EPC page that holds its SECS. The security metadata for each EPC page is held in an internal micro-architectural structure called Enclave Page Cache Map (EPCM, see Section 36.5.1).

The EPC is managed by privileged software. Intel SGX provides a set of instructions for adding and removing content to and from the EPC. The EPC may be configured by BIOS at boot time. On implementations in which EPC memory is part of system DRAM, the contents of the EPC are protected by an encryption engine.

36.5.1 Enclave Page Cache Map (EPCM)

The EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds one entry for each page in the EPC. The format of the EPCM is micro-architectural, and consequently is implementation dependent. However, the EPCM contains the following architectural information:

- The status of EPC page with respect to validity and accessibility.
- An SECS identifier (see Section 37.19) of the enclave to which the page belongs.
- The type of page: regular, SECS, TCS or VA.
- The linear address through which the enclave is allowed to access the page.
- The specified read/write/execute permissions on that page.

The EPCM structure is used by the CPU in the address-translation flow to enforce access-control on the EPC pages. The EPCM structure is described in Table 37-27, and the conceptual access-control flow is described in Section 37.5.

The EPCM entries are managed by the processor as part of various instruction flows.

36.6 ENCLAVE INSTRUCTIONS AND INTEL® SGX

The enclave instructions available with Intel SGX are organized as leaf functions under three instruction mnemonics: ENCLS (ring 0), ENCLU (ring 3), and ENCLV (VT root mode). Each leaf function uses EAX to specify the leaf function index, and may require additional implicit input registers as parameters. The use of EAX is implied implicitly by the ENCLS, ENCLU, and ENCLV instructions; ModR/M byte encoding is not used with ENCLS, ENCLU, and ENCLV. The use of additional registers does not use ModR/M encoding and is implied implicitly by the respective leaf function index.

Each leaf function index is also associated with a unique, leaf-specific mnemonic. A long-form expression of Intel SGX instruction takes the form of ENCLx[LEAF_MNEMONIC], where 'x' is either 'S', 'U', or 'V'. The long-form expression provides clear association of the privilege-level requirement of a given "leaf mnemonic". For simplicity, the unique "Leaf_Mnemonic" name is used (omitting the ENCLx for convenience) throughout in this document.

Details of individual SGX leaf functions are described in Chapter 40. Table 36-1 provides a summary of the instruction leaves that are available in the initial implementation of Intel SGX, which is introduced in the 6th generation Intel Core processors. Table 36-2 summarizes enhancement of Intel SGX for future Intel processors.

Table 36-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add an EPC page to an enclave.	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[EDBGRD]	Read data from a debug enclave by debugger.	ENCLU[EREPORT]	Create a cryptographic report.

Table 36-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EDBGWR]	Write data into a debug enclave 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 in blocked state.		
ENCLS[ELDU]	Load an EPC page in unblocked state.		
ENCLS[EPA]	Add an EPC page to create a version array.		
ENCLS[EREMOVE]	Remove an EPC page from an enclave.		
ENCLS[ETRACK]	Activate EBLOCK checks.		
ENCLS[EWB]	Write back/invalidate an EPC page.		

Table 36-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EAUG]	Allocate EPC page to an existing enclave.	ENCLU[EACCEPT]	Accept EPC page into the enclave.
ENCLS[EMODPR]	Restrict page permissions.	ENCLU[EMODPE]	Enhance page permissions.
ENCLS[EMODT]	Modify EPC page type.	ENCLU[EACCEPTCOPY]	Copy contents to an augmented EPC page and accept the EPC page into the enclave.

Table 36-3. VMX Operation and Supervisor Mode Enclave Instruction Leaf Functions in Long-Form of OVERSUB

Supervisor Instruction	Description	User Instruction	Description
ENCLV[EDECVIRTCHILD]	Decrement the virtual child page count.	ENCLS[ERDINFO]	Read information about EPC page.
ENCLV[EINCVIRTCHILD]	Increment the virtual child page count.	ENCLS[TRACKC]	Activate EBLOCK checks with conflict reporting.
ENCLV[ESETCONTEXT]	Set virtualization context.	ENCLS[ELDBC/UC]	Load an EPC page with conflict reporting.

36.7 DISCOVERING SUPPORT FOR INTEL® SGX AND ENABLING ENCLAVE INSTRUCTIONS

Detection of support of Intel SGX and enumeration of available and enabled Intel SGX resources are queried using the CPUID instruction. The enumeration interface comprises the following:

- Processor support of Intel SGX is enumerated by a feature flag in CPUID leaf 07H: CPUID.(EAX=07H, ECX=0H):EBX.SGX[bit 2]. If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor has support for Intel SGX, and requires opt-in enabling by BIOS via IA32_FEATURE_CONTROL MSR.

  If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, CPUID will report via the available sub-leaves of CPUID.(EAX=12H) on available and/or configured Intel SGX resources.
- The available and configured Intel SGX resources enumerated by the sub-leaves of CPUID.(EAX=12H) depend on the state of BIOS configuration.

36.7.1 Intel® SGX Opt-In Configuration

On processors that support Intel SGX, IA32_FEATURE_CONTROL provides the SGX_ENABLE field (bit 18). Before system software can configure and enable Intel SGX resources, BIOS is required to set IA32_FEATURE_CONTROL.SGX_ENABLE = 1 to opt-in the use of Intel SGX by system software.

The semantics of setting SGX_ENABLE follows the rules of IA32_FEATURE_CONTROL.LOCK (bit 0). Software is considered to have opted into Intel SGX if and only if IA32_FEATURE_CONTROL.SGX_ENABLE and IA32_FEATURE_CONTROL.LOCK are set to 1. The setting of IA32_FEATURE_CONTROL.SGX_ENABLE (bit 18) is not reflected by CPUID.

Table 36-4. Intel® SGX Opt-in and Enabling Behavior

CPUID.(07H,0H):EBX.SGX	CPUID.(12H)	FEATURE_CONTROL.LOCK	FEATURE_CONTROL.SGX_ENABLE	Enclave Instruction
0	Invalid	X	X	#UD
1	Valid*	X	X	#UD**
1	Valid*	0	X	#GP
1	Valid*	1	0	#GP
1	Valid*	1	1	Available (see Table 36-5 for details of SGX1 and SGX2).
* Leaf 12H enumeration results are dependent on enablement. ** See list of conditions in the #UD section of the reference pages of ENCLS and ENCLU				

36.7.2 Intel® SGX Resource Enumeration Leaves

If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor also supports querying CPUID with EAX=12H on Intel SGX resource capability and configuration. The number of available sub-leaves in leaf 12H depends on the Opt-in and system software configuration. Information returned by CPUID.12H is thread specific; software should not assume that if Intel SGX instructions are supported on one hardware thread, they are also supported elsewhere.

A properly configured processor exposes Intel SGX functionality with CPUID.EAX=12H reporting valid information (non-zero content) in three or more sub-leaves, see Table 36-5.

- CPUID.(EAX=12H, ECX=0H) enumerates Intel SGX capability, including enclave instruction opcode support.
- CPUID.(EAX=12H, ECX=1H) enumerates Intel SGX capability of processor state configuration and enclave configuration in the SECS structure (see Table 37-3).
- CPUID.(EAX=12H, ECX >1) enumerates available EPC resources.

Table 36-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=0)		Description Behavior
Register	Bits	
EAX	0	SGX1: If 1, indicates leaf functions of SGX1 instruction listed in Table 36-1 are supported.
	1	SGX2: If 1, indicates leaf functions of SGX2 instruction listed in Table 36-2 are supported.
	4:2	Reserved (0)
	5	OVERSUB: If 1, indicates Intel SGX supports instructions: EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT.
	6	OVERSUB: If 1, indicates Intel SGX supports instructions: ETRACKC, ERDINFO, ELDBC, and ELDUC.
	31:7	Reserved (0)
EBX	31:0	MISCSELECT: Reports the bit vector of supported extended features that can be written to the MISC region of the SSA.
ECX	31:0	Reserved (0).

Table 36-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=0)		Description Behavior
Register	Bits	
EDX	7:0	MaxEnclaveSize_Not64: the maximum supported enclave size is 2^(EDX[7:0]) bytes when not in 64-bit mode.
	15:8	MaxEnclaveSize_64: the maximum supported enclave size is 2^(EDX[15:8]) bytes when operating in 64-bit mode.
	31:16	Reserved (0).

Table 36-6. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=1)		Description Behavior
Register	Bits	
EAX	31:0	Report the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE. SECS.ATTRIBUTES[n] can be set to 1 using ECREATE only if EAX[n] is 1, where n < 32.
EBX	31:0	Report the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE. SECS.ATTRIBUTES[n+32] can be set to 1 using ECREATE only if EBX[n] is 1, where n < 32.
ECX	31:0	Report the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE. SECS.ATTRIBUTES[n+64] can be set to 1 using ECREATE only if ECX[n] is 1, where n < 32.
EDX	31:0	Report the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE. SECS.ATTRIBUTES[n+96] can be set to 1 using ECREATE only if EDX[n] is 1, where n < 32.

On processors that support Intel SGX1 and SGX2, CPUID leaf 12H sub-leaf 2 report physical memory resources available for use with Intel SGX. These physical memory sections are typically allocated by BIOS as **Processor Reserved Memory**, and available to the OS to manage as EPC.

To enumerate how many EPC sections are available to the EPC manager, software can enumerate CPUID leaf 12H with sub-leaf index starting from 2, and decode the sub-leaf-type encoding (returned in EAX[3:0]) until the sub-leaf type is invalid. All invalid sub-leaves of CPUID leaf 12H return EAX/EBX/ECX/EDX with 0.

Table 36-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources

CPUID.(EAX=12H,ECX > 1)		Description Behavior
Register	Bits	
EAX	3:0	0000b: This sub-leaf is invalid; EDX:ECX:EBX:EAX return 0. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other encoding are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the physical address of the base of the EPC section.
EBX	19:0	If EAX[3:0] = 0001b, these are bits 51:32 of the physical address of the base of the EPC section.
	31:20	Reserved.
ECX	3: 0	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 encoding are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.

Table 36-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources

CPUID.(EAX=12H,ECX > 1)		Description Behavior
Register	**Bits**	
EDX	19: 0	If EAX[3:0] = 0001b, these are bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.
	31:20	Reserved.

24.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: Various updates throughout chapter regarding Intel SGX and new Intel SGX VM Over-subscription feature.

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 that the processor applies enhanced access control. This ranged 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.
- 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.
— Target page must not have a restricted page type[1] (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 only accessible 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 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.

1. EPCM may allow write, read or execute access only for pages with page type PT_REG.

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
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	

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
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 37.19.

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.
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	260	3834	The RESERVED field consists of the following: ▪ 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. The entire 3836 byte field must be cleared prior to executing ECREATE.

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 EDBGRD and EDBGWR.
MODE64BIT	2	Enclave runs in 64-bit mode.
RESERVED	3	Must be Zero.
PROVISIONKEY	4	Provisioning Key is available from EGETKEY.

Table 37-3. Layout of ATTRIBUTES Structure

Field	Bit Position	Description
EINITTOKEN_KEY	5	EINIT token key is available from EGETKEY.
RESERVED	6	Must be zero.
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.
Reserved	31:1	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 37-5. 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 processer 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.
OFSBASGX	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.
OGSBASGX	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.

Table 37-5. Layout of Thread Control Structure (TCS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
GSLIMIT	68	4	Size to become the new GS limit in 32-bit mode.
RESERVED	72	4024	Must be zero.

37.8.1 TCS.FLAGS

Table 37-6. 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 37-7). 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 37-7. Top-to-Bottom Layout of an SSA Frame

Region	Offset (Byte)	Size (Bytes)	Description
XSAVE	0	Calculate using CPUID leaf 0DH 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 0DH 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.
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 37-8 for layout of the GPRSGX region.

37.9.1 GPRSGX Region

The layout of the GPRSGX region is shown in Table 37-8.

Table 37-8. 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 37-9. The VALID bit is set only for the exceptions conditions which are reported inside an enclave. See Table 37-10 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.

Table 37-9. 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: • Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM. • Conditionally supported exception: — #PF, #GP if SECS.MISCSELECT.EXINFO = 1.

37.9.1.2 VECTOR Field Definition

Table 37-10 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 37-10. 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.

37.9.2 MISC Region

The layout of the MISC region is shown in Table 37-11. The number of components that the processor supports in the MISC region corresponds to the set 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 37-11. 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 37-11. 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 37-11. 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.
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 37-12 contains the layout of the EXINFO structure that provides additional information.

Table 37-12. 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.
ERRCD	8	4	Exception error code for either #GP or #PF.
RESERVED	12	4	

37.9.2.2 Page Fault Error Codes

Table 37-13 contains page fault error code that may be reported in EXINFO.ERRCD.

Table 37-13. Page Fault Error Codes

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 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 37-14. 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.

37.11 SECURITY INFORMATION (SECINFO)

The SECINFO data structure holds meta-data about an enclave page.

Table 37-15. 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.

37.11.1 SECINFO.FLAGS

The SECINFO.FLAGS are a set of fields describing the properties of an enclave page.

Table 37-16. 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.

37.11.2 PAGE_TYPE Field Definition

The SECINFO flags and EPC flags contain bits indicating the type of page.

Table 37-17. 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.
	All other	Reserved.

37.12 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 37-18. 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.

37.13 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 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

SIGSTRUCT must be page aligned

In column 5 of Table 37-19, 'Y' indicates that this field should be included in the signature generated by the developer.

Table 37-19. Layout of Enclave Signature Structure (SIGSTRUCT)

Field	OFFSET (Bytes)	Size (Bytes)	Description	Signed
HEADER	0	16	Must be byte stream 06000000E10000000000010000000000H	Y
VENDOR	16	4	Intel Enclave: 00008086H Non-Intel Enclave: 00000000H	Y
DATE	20	4	Build date is yyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31	Y
HEADER2	24	16	Must be byte stream 01010000600000006000000001000000H	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
RESERVED	908	4	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

* 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.

37.14 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 37-20. 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.
RESERVED	212	24	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.

37.15 REPORT (REPORT)

The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

Table 37-21. 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.
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.
CONFIGID	192	64	Value provided by SW to identify enclave's post EINIT configuration.
ISVPRODID	256	02	Product ID of enclave.
ISVSVN	258	02	Security version number (SVN) of the enclave.
CONFIGSVN	260	02	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.

Table 37-21. Layout of REPORT

Field	OFFSET (Bytes)	Size (Bytes)	Description
REPORTDATA	320	64	Data provided by the user and protected by the REPORT's MAC, see Section 37.15.1.
KEYID	384	32	Value for key wear-out protection.
MAC	416	16	Message Authentication Code on the report using report key.

37.15.1 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.

37.16 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 37-22. 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.
RESERVED	48	2	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.

37.17 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 37-23. Layout of KEYREQUEST Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
KEYNAME	0	02	Identifies the Key Required.
KEYPOLICY	02	02	Identifies which inputs are required to be used in the key derivation.
ISVSVN	04	02	The ISV security version number that will be used in the key derivation.
RESERVED	06	02	Must be zero.
CPUSVN	08	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	04	A mask defining which MISCSELECT bits will be included in key derivation.

Table 37-23. Layout of KEYREQUEST Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
CONFIGSVN	76	02	Identifies which enclave Configuration's Security Version should be used in key derivation.
RESERVED	78	434	

37.17.1 KEY REQUEST KeyNames

Table 37-24. Supported KEYName Values

Key Name	Value	Description
EINITTOKEN_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 other	Reserved

37.17.2 Key Request Policy Structure

Table 37-25. 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' 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.

37.18 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 37-26. Layout of Version Array Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
Slot 0	0	08	Version Slot 0
Slot 1	8	08	Version Slot 1
…			
Slot 511	4088	08	Version Slot 511

37.19 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 37-27. 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).
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.

37.20 READ INFO (RDINFO)

The RDINFO structure contains status information about an EPC page. It must be aligned to 32-Bytes.

Table 37-28. 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.

37.20.1 RDINFO Status Structure

Table 37-29. 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	

37.20.2 RDINFO Flags Structure

Table 37-30. 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.

25.Updates to Chapter 38, Volume 3D

Change bars show changes to Chapter 38 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Various updates throughout chapter regarding Intel SGX and new Intel SGX VM Over-subscription feature.

CHAPTER 38
ENCLAVE OPERATION

The following aspects of enclave operation are described in this chapter:

- Enclave creation: Includes loading code and data from outside of enclave into the EPC and establishing the enclave entity.
- Adding pages and measuring the enclave.
- Initialization of an enclave: Finalizes the cryptographic log and establishes the enclave identity and sealing identity.
- Enclave entry and exiting including:
  - — Controlled entry and exit.
  - — Asynchronous Enclave Exit (AEX) and resuming execution after an AEX.

38.1 CONSTRUCTING AN ENCLAVE

Figure 38-1 illustrates a typical Enclave memory layout.

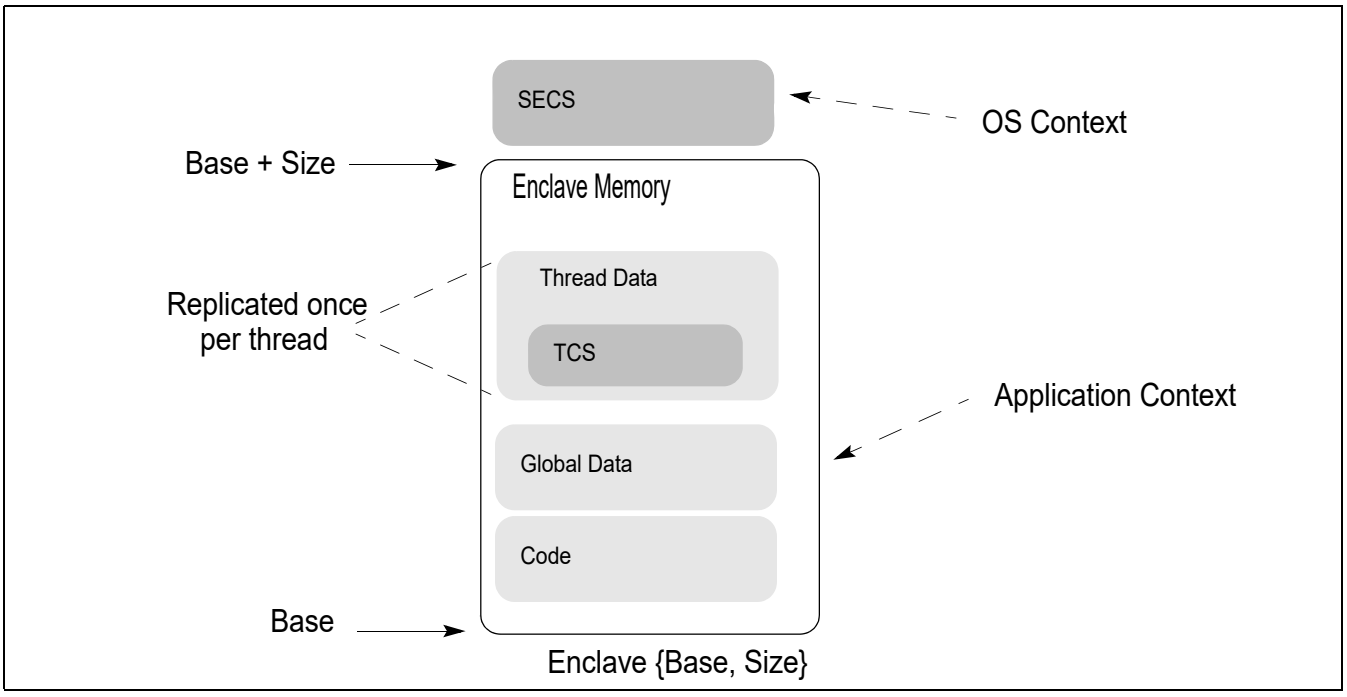


Figure 38-1. Enclave Memory Layout

The enclave creation, commitment of memory resources, and finalizing the enclave's identity with measurement comprises multiple phases. This process can be illustrated by the following exemplary steps:

1. The application hands over the enclave content along with additional information required by the enclave creation API to the enclave creation service running at privilege level 0.
2. The enclave creation service running at privilege level 0 uses the ECREATE leaf function to set up the initial environment, specifying base address and size of the enclave. This address range, the ELRANGE, is part of the application's address space. This reserves the memory range. The enclave will now reside in this address

region. ECREATE also allocates an Enclave Page Cache (EPC) page for the SGX Enclave Control Structure (SECS). Note that this page is not required to be a part of the enclave linear address space and is not required to be mapped into the process.

3. The enclave creation service uses the EADD leaf function to commit EPC pages to the enclave, and use EEXTEND to measure the committed memory content of the enclave. For each page to be added to the enclave:
   — Use EADD to add the new page to the enclave.
   — If the enclave developer requires measurement of the page as a proof for the content, use EEXTEND to add a measurement for 256 bytes of the page. Repeat this operation until the entire page is measured.
4. The enclave creation service uses the EINIT leaf function to complete the enclave creation process and finalize the enclave measurement to establish the enclave identity. Until an EINIT is executed, the enclave is not permitted to execute any enclave code (i.e. entering the enclave by executing EENTER would result in a fault).

38.1.1 ECREATE

The ECREATE leaf function sets up the initial environment for the enclave by reading an SGX Enclave Control Structure (SECS) that contains the enclave's address range (ELRANGE) as defined by BASEADDR and SIZE, the ATTRIBUTES and MISCSELECT bitmaps, and the SSAFRAMESIZE. It then securely stores this information in an Enclave Page Cache (EPC) page. ELRANGE is part of the application's address space. ECREATE also initializes a cryptographic log of the enclave's build process.

38.1.2 EADD and EEXTEND Interaction

Once the SECS has been created, enclave pages can be added to the enclave via EADD. This involves converting a free EPC page into either a PT_REG or a PT_TCS page.

When EADD is invoked, the processor will update the EPCM entry with the type of page (PT_REG or PT_TCS), the linear address used by the enclave to access the page, and the enclave access permissions for the page. It associates the page to the SECS provided as input. The EPCM entry information is used by hardware to manage access control to the page. EADD records EPCM information in the cryptographic log stored in the SECS and copies 4 KBytes of data from unprotected memory outside the EPC to the allocated EPC page.

System software is responsible for selecting a free EPC page. System software is also responsible for providing the type of page to be added, the attributes of the page, the contents of the page, and the SECS (enclave) to which the page is to be added as requested by the application. Incorrect data would lead to a failure of EADD or to an incorrect cryptographic log and a failure at EINIT time.

After a page has been added to an enclave, software can measure a 256 byte region as determined by the developer by invoking EEXTEND. Thus to measure an entire 4KB page, system software must execute EEXTEND 16 times. Each invocation of EEXTEND adds to the cryptographic log information about which region is being measured and the measurement of the section.

Entries in the cryptographic log define the measurement of the enclave and are critical in gaining assurance that the enclave was correctly constructed by the untrusted system software.

38.1.3 EINIT Interaction

Once system software has completed the process of adding and measuring pages, the enclave needs to be initialized by the EINIT leaf function. After an enclave is initialized, EADD and EEXTEND are disabled for that enclave (An attempt to execute EADD/EEXTEND to enclave after enclave initialization will result in a fault). The initialization process finalizes the cryptographic log and establishes the **enclave identity** and **sealing identity** used by EGETKEY and EREPORT.

A cryptographic hash of the log is stored as the **enclave identity**. Correct construction of the enclave results in the cryptographic hash matching the one built by the enclave owner and included as the ENCLAVEHASH field of SIGSTRUCT. The **enclave identity** provided by the EREPORT leaf function can be verified by a remote party.

The EINIT leaf function checks the EINIT token to validate that the enclave has been enabled on this platform. If the enclave is not correctly constructed, or the EINIT token is not valid for the platform, or SIGSTRUCT isn't properly signed, then EINIT will fail. See the EINIT leaf function for details on the error reporting.

The **enclave identity** is a cryptographic hash that reflects the enclave attributes and MISCSELECT value, content of the enclave, the order in which it was built, the addresses it occupies in memory, the security attributes, and access right permissions of each page. The **enclave identity** is established by the EINIT leaf function.

The **sealing identity** is managed by a sealing authority represented by the hash of the public key used to sign the SIGSTRUCT structure processed by EINIT. The sealing authority assigns a product ID (ISVPRODID) and security version number (ISVSVN) to a particular enclave identity.

EINIT establishes the sealing identity using the following steps:

1. Verifies that SIGSTRUCT is properly signed using the public key enclosed in the SIGSTRUCT.
2. Checks that the measurement of the enclave matches the measurement of the enclave specified in SIGSTRUCT.
3. Checks that the enclave's attributes and MISCSELECT values are compatible with those specified in SIGSTRUCT.
4. Finalizes the measurement of the enclave and records the **sealing identity** (the sealing authority, product id and security version number) and **enclave identity** in the SECS.
5. Sets the ATTRIBUTES.INIT bit for the enclave.

38.1.4 Intel® SGX Launch Control Configuration

Intel® SGX Launch Control is a set of controls that govern the creation of enclaves. Before the EINIT leaf function will successfully initialize an enclave, a designated Launch Enclave must create an EINITTOKEN for that enclave. Launch Enclaves have SECS.ATTRIBUTES.EINITTOKEN_KEY = 1, granting them access to the EINITTOKEN_KEY from the EGETKEY leaf function. EINITTOKEN_KEY must be used by the Launch Enclave when computing EINITTOKEN.MAC, the Message Authentication Code of the EINITTOKEN.

The hash of the public key used to sign the SIGSTRUCT of the Launch Enclave must equal the value in the IA32_SGXLEPUBKEYHASH MSRs. Only Launch Enclaves are allowed to launch without a valid token.

The IA32_SGXLEPUBKEYHASH MSRs are provided to designate the platform's Launch Enclave. IA32_SGXLEPUBKEYHASH defaults to digest of Intel's launch enclave signing key after reset.

IA32_FEATURE_CONTROL bit 17 controls the permissions on the IA32_SGXLEPUBKEYHASH MSRs when CPUID.(EAX=12H, ECX=00H):EAX[0] = 1. If IA32_FEATURE_CONTROL is locked with bit 17 set, IA32_SGXLEPUBKEYHASH MSRs are reconfigurable (writeable). If either IA32_FEATURE_CONTROL is not locked or bit 17 is clear, the MSRs are read only. By leaving these MSRs writable, system SW or a VMM can support a plurality of Launch Enclaves for hosting multiple execution environments. See Table 42.2.2 for more details.

38.2 ENCLAVE ENTRY AND EXITING

38.2.1 Controlled Entry and Exit

The EENTER leaf function is the method to enter the enclave under program control. To execute EENTER, software must supply an address of a TCS that is part of the enclave to be entered. The TCS holds the location inside the enclave to transfer control to and a pointer to the SSA frame inside the enclave that an AEX should store the register state to.

When a logical processor enters an enclave, the TCS is considered busy until the logical processors exits the enclave. An attempt to enter an enclave through a busy TCS results in a fault. Intel® SGX allows an enclave builder to define multiple TCSs, thereby providing support for multithreaded enclaves.

Software must also supply to EENTER the Asynchronous Exit Pointer (AEP) parameter. AEP is an address external to the enclave which an exception handler will return to using IRET. Typically the location would contain the ERESUME instruction. ERESUME transfers control back to the enclave, to the address retrieved from the enclave thread's saved state.

EENTER performs the following operations:

1. Check that TCS is not busy and flush all cached linear-to-physical mappings.
2. Change the mode of operation to be in enclave mode.
3. Save the old RSP, RBP for later restore on AEX (Software is responsible for setting up the new RSP, RBP to be used inside enclave).
4. Save XCR0 and replace it with the XFRM value for the enclave.
5. Check if software wishes to debug (applicable to a debuggable enclave):
   — If not debugging, then configure hardware so the enclave appears as a single instruction.
   — If debugging, then configure hardware to allow traps, breakpoints, and single steps inside the enclave.
6. Set the TCS as busy.
7. Transfer control from outside enclave to predetermined location inside the enclave specified by the TCS.

The EEXIT leaf function is the method of leaving the enclave under program control. EEXIT receives the target address outside of the enclave that the enclave wishes to transfer control to. It is the responsibility of enclave software to erase any secret from the registers prior to invoking EEXIT. To allow enclave software to easily perform an external function call and re-enter the enclave (using EEXIT and EENTER leaf functions), EEXIT returns the value of the AEP that was used when the enclave was entered.

EEXIT performs the following operations:

1. Clear enclave mode and flush all cached linear-to-physical mappings.
2. Mark TCS as not busy.
3. Transfer control from inside the enclave to a location on the outside specified as parameter to the EEXIT leaf function.

38.2.2 Asynchronous Enclave Exit (AEX)

Asynchronous and synchronous events, such as exceptions, interrupts, traps, SMIs, and VM exits may occur while executing inside an enclave. These events are referred to as Enclave Exiting Events (EEE). Upon an EEE, the processor state is securely saved inside the enclave (in the thread's current SSA frame) and then replaced by a synthetic state to prevent leakage of secrets. The process of securely saving state and establishing the synthetic state is called an Asynchronous Enclave Exit (AEX). Details of AEX is described in Chapter 39, "Enclave Exiting Events".

As part of most EEEs, the AEP is pushed onto the stack as the location of the eventing address. This is the location where control will return to after executing the IRET. The ERESUME leaf function can be executed from that point to reenter the enclave and resume execution from the interrupted point.

After AEX has completed, the logical processor is no longer in enclave mode and the exiting event is processed normally. Any new events that occur after the AEX has completed are treated as having occurred outside the enclave (e.g. a #PF in dispatching to an interrupt handler).

38.2.3 Resuming Execution after AEX

After system software has serviced the event that caused the logical processor to exit an enclave, the logical processor can continue enclave execution using ERESUME. ERESUME restores processor state and returns control to where execution was interrupted.

If the cause of the exit was an exception or a fault and was not resolved, the event will be triggered again if the enclave is re-entered using ERESUME. For example, if an enclave performs a divide by 0 operation, executing ERESUME will cause the enclave to attempt to re-execute the faulting instruction and result in another divide by 0 exception. Intel® SGX provides the means for an enclave developer to handle enclave exceptions from within the enclave. Software can enter the enclave at a different location and invoke the exception handler within the enclave by executing the EENTER leaf function. The exception handler within the enclave can read the fault information from the SSA frame and attempt to resolve the faulting condition or simply return and indicate to software that the enclave should be terminated (e.g. using EEXIT).

38.2.3.1 ERESUME Interaction

ERESUME restores registers depending on the mode of the enclave (32 or 64 bit).

- In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32-bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are restored from the thread's GPR area of the current SSA frame. Neither the upper 32 bits of the legacy registers nor the 64-bit registers (R8 … R15) are loaded.
- In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 … R15, RIP and RFLAGS) are loaded.

Extended features specified by SECS.ATTRIBUTES.XFRM are restored from the XSAVE area of the current SSA frame. The layout of the x87 area depends on the current values of IA32_EFER.LMA and CS.L:

- IA32_EFER.LMA = 0 || CS.L = 0
  - 32-bit load in the same format that XSAVE/FXSAVE uses with these values.
- IA32_EFER.LMA = 1 && CS.L = 1
  - 64-bit load in the same format that XSAVE/FXSAVE uses with these values as if REX.W = 1.

38.3 CALLING ENCLAVE PROCEDURES

38.3.1 Calling Convention

In standard call conventions subroutine parameters are generally pushed onto the stack. The called routine, being aware of its own stack layout, knows how to find parameters based on compile-time-computable offsets from the SP or BP register (depending on runtime conventions used by the compiler).

Because of the stack switch when calling an enclave, stack-located parameters cannot be found in this manner. Entering the enclave requires a modified parameter passing convention.

For example, the caller might push parameters onto the untrusted stack and then pass a pointer to those parameters in RAX to the enclave software. The exact choice of calling conventions is up to the writer of the edge routines; be those routines hand-coded or compiler generated.

38.3.2 Register Preservation

As with most systems, it is the responsibility of the callee to preserve all registers except that used for returning a value. This is consistent with conventional usage and tends to optimize the number of register save/restore operations that need be performed. It has the additional security result that it ensures that data is scrubbed from any registers that were used by enclave to temporarily contain secrets.

38.3.3 Returning to Caller

No registers are modified during EEXIT. It is the responsibility of software to remove secrets in registers before executing EEXIT.

38.4 INTEL® SGX KEY AND ATTESTATION

38.4.1 Enclave Measurement and Identification

During the enclave build process, two "measurements" are taken of each enclave and are stored in two 256-bit Measurement Registers (MR): MRENCLAVE and MRSIGNER. MRENCLAVE represents the enclave's contents and build process. MRSIGNER represents the entity that signed the enclave's SIGSTRUCT.

The values of the Measurement Registers are included in attestations to identify the enclave to remote parties. The MRs are also included in most keys, binding keys to enclaves with specific MRs.

38.4.1.1 MRENCLAVE

MRENCLAVE is a unique 256 bit value that identifies the code and data that was loaded into the enclave during the initial launch. It is computed as a SHA256 hash that is initialized by the ECREATE leaf function. EADD and EEXTEND leaf functions record information about each page and the content of those pages. The EINIT leaf function finalizes the hash, which is stored in SECS.MRENCLAVE. Any tampering with the build process, contents of a page, page permissions, etc will result in a different MRENCLAVE value.

Figure 38-2 illustrates a simplified flow of changes to the MRENCLAVE register when building an enclave:

- Enclave creation with ECREATE.
- Copying a non-enclave source page into the EPC of an un-initialized enclave with EADD.
- Updating twice of the MRENCLAVE after modifying the enclave's page content, i.e. EEXTEND twice.
- Finalizing the enclave build with EINIT.

Details on specific values inserted in the hash are available in the individual instruction definitions.

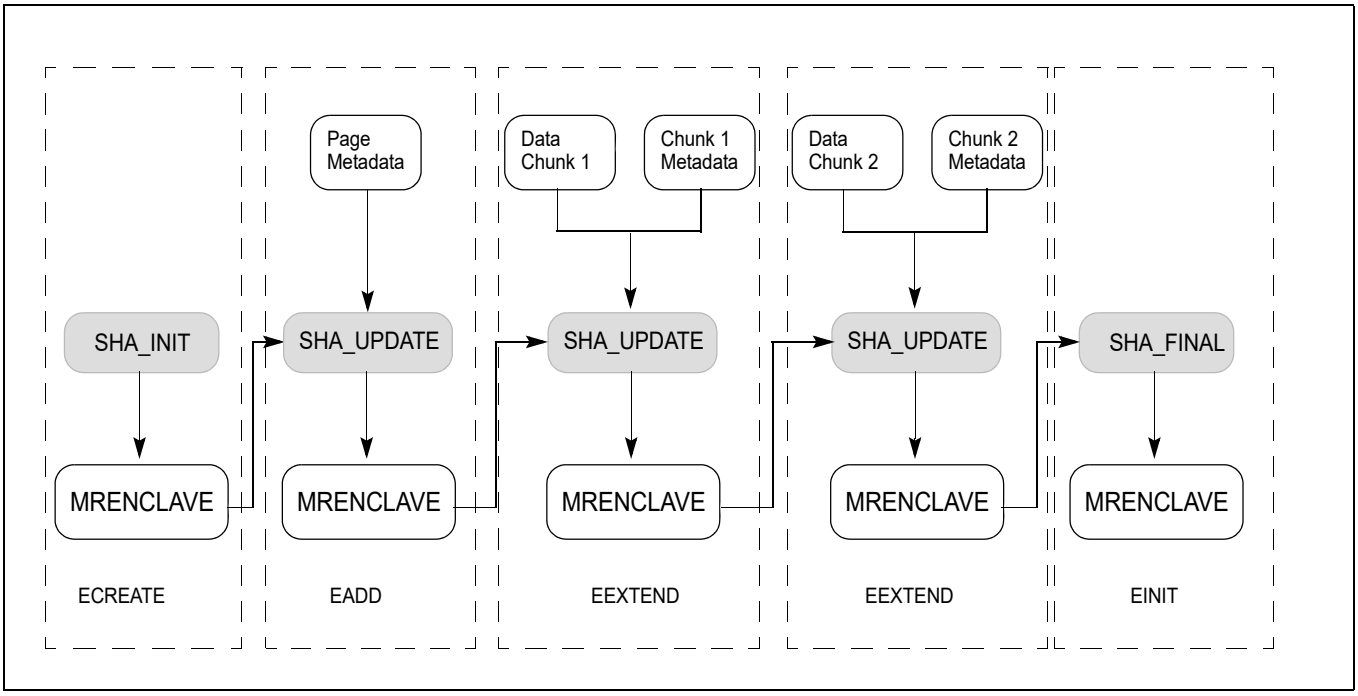


Figure 38-2. Measurement Flow of Enclave Build Process

38.4.1.2 MRSIGNER

Each enclave is signed using a 3072 bit RSA key. The signature is stored in the SIGSTRUCT. In the SIGSTRUCT, the enclave's signer also assigns a product ID (ISVPRODID) and a security version (ISVSVN) to the enclave. MRSIGNER is the SHA-256 hash of the signer's public key. For platforms that support Key Separation and Sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) the SIGSTRUCT can additionally specify an 16 byte extended product ID (ISVEXTPRODID), and a 16 byte family ID (ISVFAMILYID).

In attestation, MRSIGNER can be used to allow software to approve of an enclave based on the author rather than maintaining a list of MRENCLAVEs. It is used in key derivation to allow software to create a lineage of an application. By signing multiple enclaves with the same key, the enclaves will share the same keys and data. Combined

with security version numbering, the author can release multiple versions of an application which can access keys for previous versions, but not future versions of that application.

38.4.1.3 CONFIGID

For platforms that support enhancements for key separation and sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) when the enclave is created the platform can additionally provide 32-byte configuration identifier (CONFIGID). How this value is used is dependent on the enclave but it is intended to allow enclave creators to indicate what additional content may be accepted by the enclave post-initialization.

38.4.2 Security Version Numbers (SVN)

Intel® SGX supports a versioning system that allows the signer to identify different versions of the same software released by an author. The security version is independent of the functional version an author uses and is intended to specify security equivalence. Multiple releases with functional enhancements may all share the same SVN if they all have the same security properties or posture. Each enclave has an SVN and the underlying hardware has an SVN.

The SVNs are attested to in EREPORT and are included in the derivation of most keys, thus providing separation between data for older/newer versions.

38.4.2.1 Enclave Security Version

In the SIGSTRUCT, the MRSIGNER is associated with a 16-bit Product ID (ISVPRODID) and a 16 bit integer SVN (ISVSVN). Together they define a specific group of versions of a specific product. Most keys, including the Seal Key, can be bound to this pair.

To support upgrading from one release to another, EGETKEY will return keys corresponding to any value less than or equal to the software's ISVSVN.

38.4.2.2 Hardware Security Version

CPUSVN is a 128 bit value that reflects the microcode update version and authenticated code modules supported by the processor. Unlike ISVSVN, CPUSVN is not an integer and cannot be compared mathematically. Not all values are valid CPUSVNs.

Software must ensure that the CPUSVN provided to EGETKEY is valid. EREPORT will return the CPUSVN of the current environment. Software can execute EREPORT with TARGETINFO set to zeros to retrieve a CPUSVN from REPORTDATA. Software can access keys for a CPUSVN recorded previously, provided that each of the elements reflected in CPUSVN are the same or have been upgraded.

38.4.2.3 CONFIGID Security Version

The CONFIGID field can be used to contain the hash of a signing key for verifying the additional content. In this case, similar to the relationship between MRSIGNER and ISVSVN, CONFIGID needs a CONFIGID Security Version Number. CONFIGIDSVN can be specified at the same time as CONFIGID.

38.4.3 Keys

Intel® SGX provides software with access to keys unique to each processor and rooted in HW keys inserted into the processor during manufacturing.

Each enclave requests keys using the EGETKEY leaf function. The key is based on enclave parameters such as measurement, the enclave signing key, security attributes of the enclave, and the Hardware Security version of the processor itself. A full list of parameter options is specified in the KEYREQUEST structure, see details in Section 37.17.

By deriving keys using enclave properties, SGX guarantees that if two enclaves call EGETKEY, they will receive a unique key only accessible by the respective enclave. It also guarantees that the enclave will receive the same key

on every future execution of EGETKEY. Some parameters are optional or configurable by software. For example, a Seal key can be based on the signer of the enclave, resulting in a key available to multiple enclaves signed by the same party.

The EGETKEY leaf function provides several key types. Each key is specific to the processor, CPUSVN, and the enclave that executed EGETKEY. The EGETKEY instruction definition details how each of these keys is derived, see Table 40-64. Additionally,

- SEAL Key: The Seal key is a general purpose key for the enclave to use to protect secrets. Typical uses of the Seal key are encrypting and calculating MAC of secrets on disk. There are 2 types of Seal Key described in Section 38.4.3.1.
- REPORT Key: This key is used to compute the MAC on the REPORT structure. The EREPORT leaf function is used to compute this MAC, and destination enclave uses the Report key to verify the MAC. The software usage flow is detailed in Section 38.4.3.2.
- EINITTOKEN_KEY: This key is used by Launch Enclaves to compute the MAC on EINITTOKENs. These tokens are then verified in the EINIT leaf function. The key is only available to enclaves with ATTRIBUTE.EINITTOKEN_KEY set to 1.
- PROVISIONING Key and PROVISIONING SEAL Key: These keys are used by attestation key provisioning software to prove to remote parties that the processor is genuine and identify the currently executing TCB. These keys are only available to enclaves with ATTRIBUTE.PROVISIONKEY set to 1.

38.4.3.1 Sealing Enclave Data

Enclaves can protect persistent data using Seal keys to provide encryption and/or integrity protection. EGETKEY provides two types of Seal keys specified in KEYREQUEST.KEYPOLICY field: MRENCLAVE-based key and MRSIGNER-based key.

The MRENCLAVE-based keys are available only to enclave instances sharing the same MRENCLAVE. If a new version of the enclave is released, the Seal keys will be different. Retrieving previous data requires additional software support.

The MRSIGNER-based keys are bound to the 3 tuple (MRSIGNER, ISVPRODID, ISVSVN). These keys are available to any enclave with the same MRSIGNER and ISVPRODID and an ISVSVN equal to or greater than the key in questions. This is valuable for allowing new versions of the same software to retrieve keys created before an upgrade.

For platforms that support enhancements for key separation and sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) four additional key policies for seal key derivation are provided. These add the ISVEXTPRODID, ISVFAMILYID and CONFIGID/CONFIGSVN to the key derivation. Additionally there is a policy to remove ISVPRODID from a key derivation to create a shared between different products that share the same MRSIGNER.

38.4.3.2 Using REPORTs for Local Attestation

SGX provides a means for enclaves to securely identify one another, this is referred to as "Local Attestation". SGX provides a hardware assertion, REPORT that contains calling enclaves Attributes, Measurements and User supplied data (described in detail in Section 37.15). Figure 38-3 shows the basic flow of information.

1. The source enclave determines the identity of the target enclave to populate TARGETINFO.
2. The source enclave calls EREPORT instruction to generate a REPORT structure. The EREPORT instruction conducts the following:
   — Populates the REPORT with identify information about the calling enclave.
   — Derives the Report Key that is returned when the target enclave executes the EGETKEY. TARGETINFO provides information about the target.
   — Computes a MAC over the REPORT using derived target enclave Report Key.
3. Non-enclave software copies the REPORT from source to destination.
4. The target enclave executes the EGETKEY instruction to request its REPORT key, which is the same key used by EREPORT at the source.
5. The target enclave verifies the MAC and can then inspect the REPORT to identify the source.

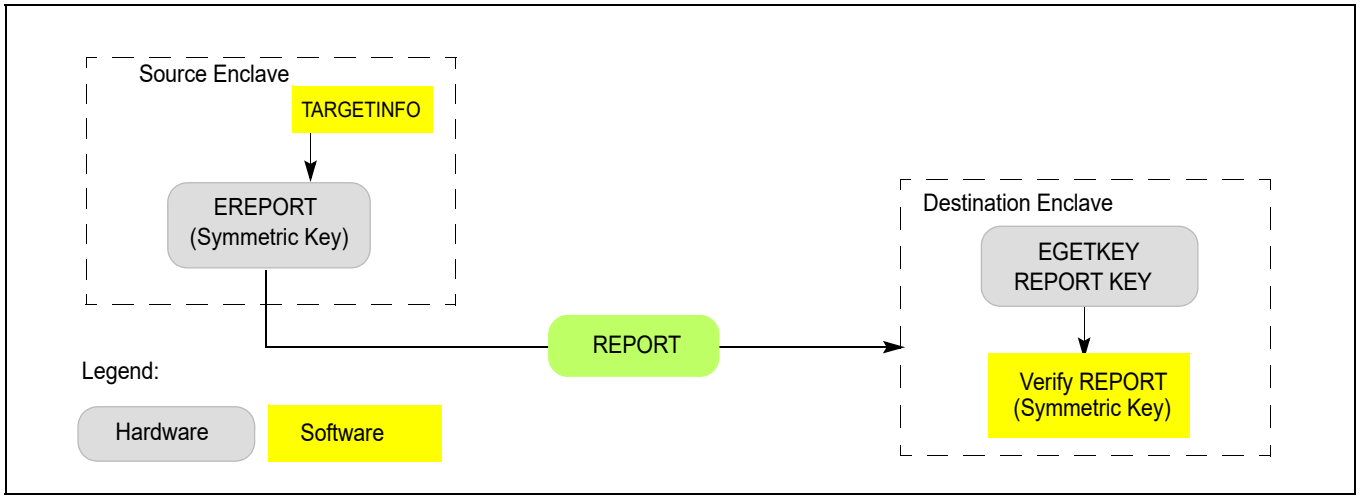


Figure 38-3. SGX Local Attestation

38.5 EPC AND MANAGEMENT OF EPC PAGES

EPC layout is implementation specific, and is enumerated through CPUID (see Table 36-7 for EPC layout). EPC is typically configured by BIOS at system boot time.

38.5.1 EPC Implementation

EPC must be properly protected against attacks. One example of EPC implementation could use a Memory Encryption Engine (MEE). An MEE provides a cost-effective mechanism of creating cryptographically protected volatile storage using platform DRAM. These units provide integrity, replay, and confidentiality protection. Details are implementation specific.

38.5.2 OS Management of EPC Pages

The EPC is a finite resource. SGX1 (i.e. CPUID.(EAX=12H, ECX=0):EAX.SGX1 = 1 but CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 0) provides the EPC manager with leaf functions to manage this resource and properly swap pages out of and into the EPC. For that, the EPC manager would need to keep track of all EPC entries, type and state, context affiliation, and SECS affiliation.

Enclave pages that are candidates for eviction should be moved to BLOCKED state using EBLOCK instruction that ensures no new cached virtual to physical address mappings can be created by attempts to reference a BLOCKED page.

Before evicting blocked pages, EPC manager should execute ETRACK leaf function on that enclave and ensure that there are no stale cached virtual to physical address mappings for the blocked pages remain on any thread on the platform.

After removing all stale translations from blocked pages, system software should use the EWB leaf function for securely evicting pages out of the EPC. EWB encrypts a page in the EPC, writes it to unprotected memory, and invalidates the copy in EPC. In addition, EWB also creates a cryptographic MAC (PCMD.MAC) of the page and stores it in unprotected memory. A page can be reloaded back to the processor only if the data and MAC match. To ensure that only the latest version of the evicted page can be loaded back, the version of the evicted page is stored securely in a Version Array (VA) in EPC.

SGX1 includes two instructions for reloading pages that have been evicted by system software: ELDU and ELDB. The difference between the two instructions is the value of the paging state at the end of the instruction. ELDU results in a page being reloaded and set to an UNBLOCKED state, while ELDB results in a page loaded to a BLOCKED state.

ELDB is intended for use by a Virtual Machine Monitor (VMM). When a VMM reloads an evicted page, it needs to restore it to the correct state of the page (BLOCKED vs. UNBLOCKED) as it existed at the time the page was evicted. Based on the state of the page at eviction, the VMM chooses either ELDB or ELDU.

38.5.2.1 Enhancement to Managing EPC Pages

On processors supporting SGX2 (i.e. CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 1), the EPC manager can manage EPC resources (while enclave is running) with more flexibility provided by the SGX2 leaf functions. The additional flexibility is described in Section 38.5.7 through Section 38.5.11.

38.5.3 Eviction of Enclave Pages

Intel SGX paging is optimized to allow the Operating System (OS) to evict multiple pages out of the EPC under a single synchronization.

The suggested flow for evicting a list of pages from the EPC is:

1. For each page to be evicted from the EPC:
   a. Select an empty slot in a Version Array (VA) page.
      - If no empty VA page slots exist, create a new VA page using the EPA leaf function.
   b. Remove linear-address to physical-address mapping from the enclave context's mapping tables (page table and EPT tables).
   c. Execute the EBLOCK leaf function for the target page. This sets the target page state to BLOCKED. At this point no new mappings of the page will be created. So any access which does not have the mapping cached in the TLB will generate a #PF.
2. For each enclave containing pages selected in step 1:
   — Execute an ETRACK leaf function pointing to that enclave's SECS. This initiates the tracking process that ensures that all caching of linear-address to physical-address translations for the blocked pages is cleared.
3. For all logical processors executing in processes (OS) or guests (VMM) that contain the enclaves selected in step 1:
   — Issue an IPI (inter-processor interrupt) to those threads. This causes those logical processors to asynchronously exit any enclaves they might be in, and as a result flush cached linear-address to physical-address translations that might hold stale translations to blocked pages. There is no need for additional measures such as performing a "TLB shootdown".
4. After enclaves exit, allow logical processors to resume normal operation, including enclave re-entry as the tracking logic keeps track of the activity.
5. For each page to be evicted:
   — Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

At this point, system software has the only copy of each page data encrypted with its page metadata in main memory.

38.5.4 Loading an Enclave Page

To reload a previously evicted page, system software needs four elements: the VA slot used when the page was evicted, a buffer containing the encrypted page contents, a buffer containing the page metadata, and the parent SECS to associate this page with. If the VA page or the parent SECS are not already in the EPC, they must be reloaded first.

1. Execute ELDB/ELDU (depending on the desired BLOCKED state for the page), passing as parameters: the EPC page linear address, the VA slot, the encrypted page, and the page metadata.

2. Create a mapping in the enclave context's mapping tables (page tables and EPT tables) to allow the application to access that page (OS: system page table; VMM: EPT).

The ELDB/ELDU instruction marks the VA slot empty so that the page cannot be replayed at a later date.

38.5.5 Eviction of an SECS Page

The eviction of an SECS page is similar to the eviction of an enclave page. The only difference is that an SECS page cannot be evicted until all other pages belonging to the enclave have been evicted. Since all other pages have been evicted, there will be no threads executing inside the enclave and tracking with ETRACK isn't necessary. When reloading an enclave, the SECS page must be reloaded before all other constituent pages.

1. Ensure all pages are evicted from enclave.
2. Select an empty slot in a Version Array page.
   — If no VA page exists with an empty slot, create a new one using the EPA function leaf.
3. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

38.5.6 Eviction of a Version Array Page

VA pages do not belong to any enclave and tracking with ETRACK isn't necessary. When evicting the VA page, a slot in a different VA page must be specified in order to provide versioning of the evicted VA page.

1. Select a slot in a Version Array page other than the page being evicted.
   — If no VA page exists with an empty slot, create a new one using the EPA leaf function.
2. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

38.5.7 Allocating a Regular Page

On processors that support SGX2, allocating a new page to an already initialized enclave is accomplished by invoking the EAUG leaf function. Typically, the enclave requests that the OS allocate a new page at a particular location within the enclave's address space. Once allocated, the page remains in a pending state until the enclave executes the corresponding EACCEPT leaf function to accept the new page into the enclave. Page allocation operations may be batched to improve efficiency.

The typical process for allocating a regular page is as follows:

1. Enclave requests additional memory from OS when the current allocation becomes insufficient.
2. The OS invokes the EAUG leaf function to add a new memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
   c. All dynamically created pages have the type PT_REG and content of all zeros.
3. The OS maps the page in the enclave context's mapping tables.
4. The enclave issues an EACCEPT instruction, which verifies the page's attributes and clears the PENDING state. At that point the page becomes accessible for normal enclave use.

38.5.8 Allocating a TCS Page

On processors that support SGX2, allocating a new TCS page to an already initialized enclave is a two-step process. First the OS allocates a regular page with a call to EAUG. This page must then be accepted and initialized by the enclave to which it belongs. Once the page has been initialized with appropriate values for a TCS page, the enclave requests the OS to change the page's type to PT_TCS. This change must also be accepted. As with allocating a regular page, TCS allocation operations may be batched.

A typical process for allocating a TCS page is as follows:

1. Enclave requests an additional page from the OS.
2. The OS invokes EAUG to add a new regular memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
3. The OS maps the page in the enclave context's mapping tables.
4. The enclave issues an EACCEPT instruction, at which point the page becomes accessible for normal enclave use.
5. The enclave initializes the contents of the new page.
6. The enclave requests that the OS convert the page from type PT_REG to PT_TCS.
7. OS issues an EMODT instruction on the page.
   a. The parameters to EMODT indicate that the regular page should be converted into a TCS.
   b. EMODT forces all access rights to a page to be removed because TCS pages may not be accessed by enclave code.
8. The enclave issues an EACCEPT instruction to confirm the requested modification.

38.5.9 Trimming a Page

On processors that support SGX2, Intel SGX supports the trimming of an enclave page as a special case of EMODT. Trimming allows an enclave to actively participate in the process of removing a page from the enclave (deallocation) by splitting the process into first removing it from the enclave's access and then removing it from the EPC using the EREMOVE leaf function. The page type PT_TRIM indicates that a page has been trimmed from the enclave's address space and that the page is no longer accessible to enclave software. Modifications to a page in the PT_TRIM state are not permitted; the page must be removed and then reallocated by the OS before the enclave may use the page again. Page deallocation operations may be batched to improve efficiency.

The typical process for trimming a page from an enclave is as follows:

1. Enclave signals OS that a particular page is no longer in use.
2. OS invokes the EMODT leaf function on the page, requesting that the page's type be changed to PT_TRIM.
   a. SECS and VA pages cannot be trimmed in this way, so the initial type of the page must be PT_REG or PT_TCS.
   b. EMODT may only be called on valid enclave pages.
3. OS invokes the ETRACK leaf function on the enclave containing the page to track removal the TLB addresses from all the processors.
4. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
5. Enclave issues an EACCEPT leaf function.
6. The OS may now permanently remove the page from the EPC (by issuing EREMOVE).

38.5.10 Restricting the EPCM Permissions of a Page

On processors that support SGX2, restricting the EPCM permissions associated with an enclave page is accomplished using the EMODPR leaf function. This operation requires the cooperation of the OS to flush stale entries to

the page and to update the page-table permissions of the page to match. Permissions restriction operations may be batched.

The typical process for restricting the permissions of an enclave page is as follows:

1. Enclave requests that the OS to restrict the permissions of an EPC page.
2. OS performs permission restriction, flushing cached linear-address to physical-address translations, and page-table modifications.
   a. Invokes the EMODPR leaf function to restrict permissions (EMODPR may only be called on VALID pages).
   b. Invokes the ETRACK leaf function on the enclave containing the page to track removal of the TLB addresses from all the processor.
   c. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
   d. Sends IPIs to trigger enclave thread exit and TLB shootdown.
   e. OS informs the Enclave that all logical processors should now see the new restricted permissions.
3. Enclave invokes the EACCEPT leaf function.
   a. Enclave may access the page throughout the entire process.
   b. Successful call to EACCEPT guarantees that no stale cached linear-address to physical-address translations are present.

38.5.11 Extending the EPCM Permissions of a Page

On processors that support SGX2, extending the EPCM permissions associated with an enclave page is accomplished directly be the enclave using the EMODPE leaf function. After performing the EPCM permission extension, the enclave requests the OS to update the page table permissions to match the extended permission. Security wise, permission extension does not require enclave threads to leave the enclave as TLBs with stale references to the more restrictive permissions will be flushed on demand, but to allow forward progress, an OS needs to be aware that an application might signal a page fault.

The typical process for extending the permissions of an enclave page is as follows:

1. Enclave invokes EMODPE to extend the EPCM permissions associated with an EPC page (EMODPE may only be called on VALID pages).
2. Enclave requests that OS update the page tables to match the new EPCM permissions.
3. Enclave code resumes.
   a. If cached linear-address to physical-address translations are present to the more restrictive permissions, the enclave thread will page fault. The SGX2-aware OS will see that the page tables permit the access and resume the thread, which can now successfully access the page because exiting cleared the TLB.
   b. If cached linear-address to physical-address translations are not present, access to the page with the new permissions will succeed without an enclave exit.

38.5.12 VMM Oversubscription of EPC

On processors supporting oversubscription enhancements (.e. CPUID.(EAX=12H, ECX=0):EAX.[5]=1 & EAX[6] = 1) a Virtual Machine Monitor or other executive can more efficiently manage the EPC space available on the platform between virtualized entities. A typical process for using these instructions to support oversubscribing the physical EPC space on the platform is as follows:

1. VMM creates data structures for SECS tracking including a count of child pages.
2. VMM selects possible EPC victim pages.
3. VMM ages the victim pages. Some of the selected pages will be accessed by the guest. In this case the VMM will remove these pages from the victim pool and return them to the guest.
4. VMM makes remaining pages not present in EPT. It then issues IPI on each page to remove TLB mappings.

5. For every EPC victim page the VMM obtains the victim's SECS page info using ERDINFO.
   a. ENCLAVECONTEXT field in RDINFO structure will indicate the location of SECS, and the PAGE_TYPE field will indicate the page type.
   b. Child pages of SECS can be evicted.
   c. SECS pages may be evicted if the child count is zero.
   d. Some pages may be returned to active state depending on such things as page type or child count.
6. VMM increments its evicted page count for the SECS of each page (stored in the data structure created in 1).
7. If this is the first evicted page of that SECS, set Marker on SECS of the victim page (EINCVIRTCHILD). This locks the SECS in the guest. The guest cannot page out the SECS.
8. EBLOCK, ETRACK, EWB eviction sequence is executed for page.
9. After loading an SECS page back in, the VMM will set the correct ENCLAVECONTEXT for the guest using ESETCONTEXT instruction.

38.6 CHANGES TO INSTRUCTION BEHAVIOR INSIDE AN ENCLAVE

This section covers instructions whose behavior changes when executed in enclave mode.

38.6.1 Illegal Instructions

The instructions listed in Table 38-1 are ring 3 instructions which become illegal when executed inside an enclave. Executing these instructions inside an enclave will generate an exception.

The first row of Table 38-1 enumerates instructions that may cause a VM exit for VMM emulation. Since a VMM cannot emulate enclave execution, execution of any these instructions inside an enclave results in an invalid-opcode exception (#UD) and no VM exit.

The second row of Table 38-1 enumerates I/O instructions that may cause a fault or a VM exit for emulation. Again, enclave execution cannot be emulated, so execution of any these instructions inside an enclave results in #UD.

The third row of Table 38-1 enumerates instructions that load descriptors from the GDT or the LDT or that change privilege level. The former class is disallowed because enclave software should not depend on the contents of the descriptor tables and the latter because enclave execution must be entirely with CPL = 3. Again, execution of any these instructions inside an enclave results in #UD.

The fourth row of Table 38-1 enumerates instructions that provide access to kernel information from user mode and can be used to aid kernel exploits from within enclave. Execution of any these instructions inside an enclave results in #UD

Table 38-1. Illegal Instructions Inside an Enclave

Instructions	Result	Comment
CPUID, GETSEC, RDPMC, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC	#UD	Might cause VM exit.
IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD	#UD	I/O fault may not safely recover. May require emulation.
Far call, Far jump, Far Ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/FS/GS, SYSCALL, SYSENTER	#UD	Access segment register could change privilege level.
SMSW	#UD	Might provide access to kernel information.
ENCLU[EENTER], ENCLU[ERESUME]	#GP	Cannot enter an enclave from within an enclave.

RDTSC and RDTSCP are legal inside an enclave for processors that support SGX2 (subject to the value of CR4.TSD). For processors which support SGX1 but not SGX2, RDTSC and RDTSCP will cause #UD.

RDTSC and RDTSCP instructions may cause a VM exit when inside an enclave.

Software developers must take into account that the RDTSC/RDTSCP results are not immune to influences by other software, e.g. the TSC can be manipulated by software outside the enclave.

38.6.2 RDRAND and RDSEED Instructions

These instructions may cause a VM exit if the "RDRAND exiting" VM-execution control is 1. Unlike other instructions that can cause VM exits, these instructions are legal inside an enclave. As noted in Section 6.5.5, any VM exit originating on an instruction boundary inside an enclave sets bit 27 of the exit-reason field of the VMCS. If a VMM receives a VM exit due to an attempt to execute either of these instructions determines (by that bit) that the execution was inside an enclave, it can do either of two things. It can clear the "RDRAND exiting" VM-execution control and execute VMRESUME; this will result in the enclave executing RDRAND or RDSEED again, and this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

NOTE

It is expected that VMMs that virtualize Intel SGX will not set "RDRAND exiting" to 1.

38.6.3 PAUSE Instruction

The PAUSE instruction may cause a VM exit from an enclave if the "PAUSE exiting" VM-execution control is 1. Unlike other instructions that can cause VM exits, the PAUSE instruction is legal inside an enclave. If a VMM receives a VM exit due to the 1-setting of "PAUSE exiting", it can do either of two things. It can clear the "PAUSE exiting" VM-execution control and execute VMRESUME; this will result in the enclave executing PAUSE again, but this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

The PAUSE instruction may also cause a VM exit outside of an enclave if the "PAUSE-loop exiting" VM-execution control is 1, but as the "PAUSE-loop exiting" control is ignored at CPL > 0 (see Section 25.1.3), VM exit from an enclave due to the 1-setting of "PAUSE-LOOP exiting" will never occur.

NOTE

It is expected that VMMs that virtualize Intel SGX will not set "PAUSE exiting" to 1.

38.6.4 INT 3 Behavior Inside an Enclave

INT3 is legal inside an enclave, however, the behavior inside an enclave is different from its behavior outside an enclave. See Section 42.4.1 for details.

38.6.5 INVD Handling when Enclaves Are Enabled

Once processor reserved memory protections are activated (see Section 38.5), any execution of INVD will result in a #GP(0).

26.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: Various updates throughout chapter regarding Intel SGX and new Intel SGX VM Over-subscription feature.

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	RDX
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	RDX
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_BLSTATE	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, EINCVIRTCHILD
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_ELRANGE	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_XCR0	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

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 leafs 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
EDBGRD	Target	[DS:RCX]	Shared	#GP	
EDBGWR	Target	[DS:RCX]	Shared	#GP	
EDECVIRTCHILD	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	
EDBGRD	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_CONFLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	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	NP	V/V	SGX1	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.3

Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the "enable ENCLS exiting" VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the "enable ENCLS exiting" VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation and the "enable ENCLS exiting" VM-execution control is 1
    THEN
        IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLS_exiting_bitmap[63] = 1
            THEN VM exit;
        FI;
FI;
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number)
    THEN #GP(0); FI;
```

```
IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	NP	V/V	SGX1	This instruction is used to execute non-privileged Intel SGX leaf functions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.4

Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE← 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number
    THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
    THEN #GP(0); FI;

IN_64BIT_MODE ← IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and (CS.D = 0 or DS.B = 0)
```

```
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, or EACCEPTCOPY *)
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If operating in 16-bit mode.
	If data segment is in 16-bit mode.
	If CR0.PG = 0 or CR0.NE= 0.
#NM	If CR0.TS = 1.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If CR0.NE= 0.
#NM	If CR0.TS = 1.

ENCLV—Execute an Enclave VMM Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C0 ENCLV	NP	V/V	SGX1	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
NP	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 ENCLS 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 in VMX non-root operation and IA_32_EFER.LMA = 1 and CS.L = 1
    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 invalid leaf number)
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions.

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.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/OSIZE/REP/VEX prefix is 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 mis-configured 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) )
    THEN #GP(0); FI;

(* Check the EPC page for concurrency *)
IF (SECS 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.ATTIBUTES.MODE64BIT = 0) and
            ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
        BREAK;
    PT_REG:
        IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #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.
#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.
#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 SECINFO (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_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) or (DS:RBX.SECINFO 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 #GP(0); FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); 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;

(* Clear the content of EPC page*)
DS:RCX[32767:0] ← 0;

(* Set EPCM security attributes *)
EPCM(DS:RCX).R ← 1;
EPCM(DS:RCX).W ← 1;
EPCM(DS:RCX).X ← 0;
EPCM(DS:RCX).PT ← PT_REG;
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) )
    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_SRCPGE[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;

(* 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*)
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 0FFFFFFFF00000000H) )
    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;
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)	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)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

64-Bit Mode Exceptions

#GP(0)	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.
#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)	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 EDBGRD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGRD	Target [DS:RCX]	Shared	#GP	

Table 40-19. Additional Concurrency Restrictions of EDBGRD

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
EDBGRD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGRD 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 EPCM *)
IF (Other EPCM modifying instructions 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 *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_VA))
    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] ← 0FFFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] ← 0H;
                FI;
            ELSE
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN EBX[31:0] ← 0FFFFFFFFH;
                    ELSE EBX[31:0] ← 0H;
                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

None

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.

#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)	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 EDBGRD 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 EPCM *)
IF (Other EPCM modifying instructions 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 *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) )
    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:RCS).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 & 0FF8H) )
    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

None

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 or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#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 or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#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 compromising 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_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.

```
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 *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) )
    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 ATTRIBUETS.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)	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)	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)	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)	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 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / 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.

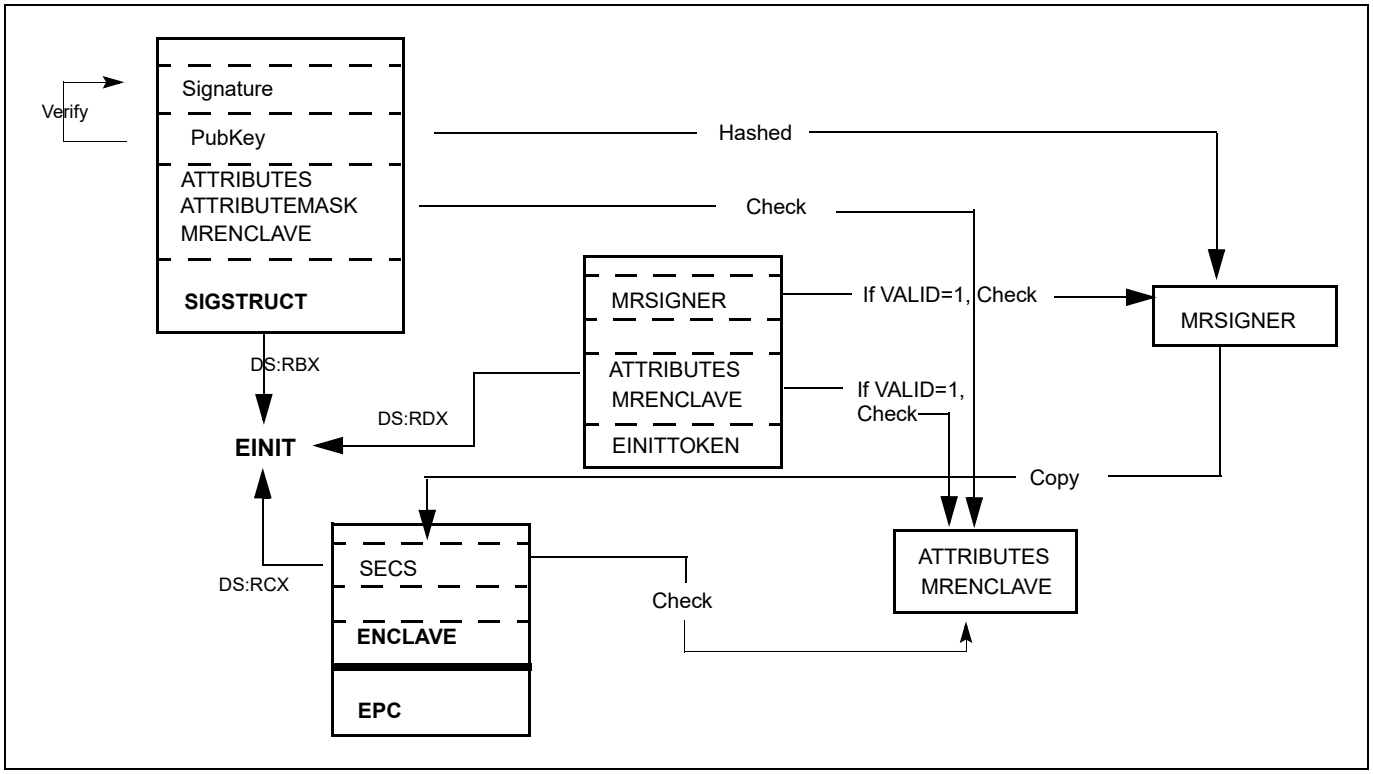


Figure 40-1. Relationships Between SECS, SIGSTRUCT and EINITTOKEN

EINIT Memory Parameter Semantics

SIGSTRUCT	SECS	EINITTOKEN
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.ATTRIBUTEMASK with SECS.ATTRIBUTES.

If EINITTOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

If EINITTOKEN.VALID is 1, checks the validity of EINITTOKEN.

If EINITTOKEN.VALID is 1, checks that EINITTOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITTOKEN.VALID is 1 and EINITTOKEN.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 EINITTOKEN 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 ≠ 06000000E10000000000010000000000h) or
     ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
     (TMP_SIG HEADER2 ≠ 01010000600000006000000001000000h) 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 ATTRIBUETS.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 ← 0000000000000020H;
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 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 be ≤ CR_CPUSVN *)
IF (TMP_TOKEN.CPUSVN > 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.ISVSVN;
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.CPUSVN;
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;

(* 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;

(* 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/ELBUC—Load an EPC Page and Marked 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[5]	This leaf function behaves lie ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELDBC]	IR	V/V	EAX[5]	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 permitted 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;
```

```
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;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
     THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~0FFFH).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) )
     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;
```

```
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) )
    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)	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)	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)	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)	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 mis-configured 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;
```

```
        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 mis-configured 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 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)	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 EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	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 another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

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)	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;
```

```
    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)
   ) 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)	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;
```

```
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) )
    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) 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. If the memory operand is not properly aligned.
#PF(error code)	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. If a memory operand is not properly aligned.
#PF(error code)	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 (DSTMP_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 & ~0FFFH).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 ) )
    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;

(* 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 ))
    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;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER)-1 : 0] ← 0;

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:RBX.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)	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)	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)	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)	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 mis-configured 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 (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

(* 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)) 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) )
    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 *)
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)
    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 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 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;
```

```
(* Copy 64 bytes of contents *)
SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured 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:RDX).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 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 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 FS 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;
```

```
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 & FFFFFFFFFFFFFFFEH) ≠ 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.ATTRIBUES.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)
```

```
            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;

CR_ENCLAVE_MODE ← 1;
CR_ACTIVE_SECS ← TMP_SECS;
CR_ELRANGE ← (TMPSECS.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;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← 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 ← 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_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;

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)	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 = 1and 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 = 1and 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 (In)

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 has the effect of abort page semantics on the next destination.

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 XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
    XCR0 ← CR_SAVE__XCR0;
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;

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)	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.

EEGETKEY 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 37.17.1) 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.

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 hardcode 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] ←

```
    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).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

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?*)
        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.ATTRIBUTEMASK;
            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;
            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;
        BREAK;
    EINITTOKEN_KEY:
        (* Check ENCLAVE has LAUNCH capability *)
        IF (TMP_CURRENTSECS.ATTRIBUTES.LAUNCHKEY = 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;
        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.ATTRIBUTEMASK;
        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;
        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)
        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 ← 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.ATTRIBUTEMASK;
            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;
            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 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 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 mis-configured 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 mis-configured 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 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 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 & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
```

```
    THEN #PF(DS:RBX);
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 #P(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:RCX).PENDING = 1) or
    (EPCM(DS:RCX).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;
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;

(* 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.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
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;

(* Calculate the derived key*)
TMP_REPORTKEY ← derive_key(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 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 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 pended 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 FS 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;
```

```
(* 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 & & FFFFFFFFFFFFFFFEH) ≠ 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 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:TMPSSA_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;
```

```
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;

(* 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_ELRANGE ← (TMP_SECS.BASEADDR, TMP_SECS.SIZE);

(* Save sate 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.IF; FI;

IF (TCS.FLAGS.OPTIN = 0)
    THEN RFLAGS.TF = 0; FI;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← 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
        Clear all pending debug exceptions;
        Clear pending MTF VM exits;
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 = 1and 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 = 1and 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)	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) )
      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
    (* EDECVIRTCHILD 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 EDECVIRTCHILD 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)	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 EINCVIRTCHILD

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
EINCVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EINCVIRTCHILD 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.

EINCVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EINCVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_INVALID_COUNTER		Attempt to increment counter that will produce an overflow.

```
(* 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) )
      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 and check for overflow *)
Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an overflow) THEN
    Locked_Decrement(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 EINCVIRTCHILD 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.

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)	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);
   goto DONE;
FI;
```

```
(* check EPC page is an SECS page *)
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN
   #PF(DS:RCX, PFEC.SGX);
   goto DONE;
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.

27. Updates to Chapter 41, Volume 3D

Change bars show changes to Chapter 41 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Various updates throughout chapter regarding Intel SGX and new Intel SGX VM Over-subscription feature.

CHAPTER 41
INTEL® SGX INTERACTIONS WITH IA32 AND INTEL® 64 ARCHITECTURE

Intel® SGX provides Intel® Architecture with a collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel® 64 architectures. These Intel SGX instructions are designed to work with legacy software and the various IA32 and Intel 64 modes of operation.

41.1 INTEL® SGX AVAILABILITY IN VARIOUS PROCESSOR MODES

The Intel SGX extensions (see Table 36-1) are available only when the processor is executing in protected mode of operation. Additionally, the extensions are not available in System Management Mode (SMM) of operation or in Virtual 8086 (VM86) mode of operation. Finally, all leaf functions of ENCLU and ENCLS require CR0.PG enabled.

The exact details of exceptions resulting from illegal modes and their priority are listed in the reference pages of ENCLS and ENCLU.

41.2 IA32_FEATURE_CONTROL

IA32_FEATURE_CONTROL MSR provides two new bits related to two aspects of Intel SGX: using the instruction extensions and launch control configuration.

41.2.1 Availability of Intel SGX

IA32_FEATURE_CONTROL[bit 18] allows BIOS to control the availability of Intel SGX extensions. For Intel SGX extensions to be available on a logical processor, bit 18 in the IA32_FEATURE_CONTROL MSR on that logical processor must be set, and IA32_FEATURE_CONTROL MSR on that logical processor must be locked (bit 0 must be set). See Section 36.7.1 for additional details. OS is expected to examine the value of bit 18 prior to enabling Intel SGX on the thread, as the settings of bit 18 is not reflected by CPUID.

41.2.2 Intel SGX Launch Control Configuration

The IA32_SGXLEPUBKEYHASHn MSRs used to configure authorized launch enclaves' MRSIGNER digest value. They are present on logical processors that support the collection of SGX1 leaf functions (i.e. CPUID.(EAX=12H, ECX=00H):EAX[0] = 1) and that CPUID.(EAX=07H, ECX=00H):ECX[30] = 1. IA32_FEATURE_CONTROL[bit 17] allows to BIOS to enable write access to these MSRs. If IA32_FEATURE_CONTROL.LE_WR (bit 17) is set to 1 and IA32_FEATURE_CONTROL is locked on that logical processor, IA32_SGXLEPUBKEYHASH MSRs on that logical processor are writeable. If this bit 17 is not set or IA32_FEATURE_CONTROL is not locked, IA32_SGXLEPUBKEYHASH MSRs are read only. See Section 38.1.4 for additional details.

41.3 INTERACTIONS WITH SEGMENTATION

41.3.1 Scope of Interaction

Intel SGX extensions are available only when the processor is executing in a protected mode operation (see Section 41.1 for Intel SGX availability in various processor modes). Enclaves abide by all the segmentation policies set up by the OS, but they can be more restrictive than the OS.

Intel SGX interacts with segmentation at two levels:

- The Intel SGX instruction (see the enclave instruction in Table 36-1).

- While executing inside an enclave (legacy instructions and enclave instructions permitted inside an enclave).

41.3.2 Interactions of Intel® SGX Instructions with Segment, Operand, and Addressing Prefixes

All the memory operands used by the Intel SGX instructions are interpreted as offsets within the data segment (DS). The segment-override prefix on Intel SGX instructions is ignored.

Operand size is fixed for each enclave instruction. The operand-size prefix is reserved, and results in a #UD exception if used.

All address sizes are determined by the operating mode of the processor. The address-size prefix is ignored. This implies that while operating in 64-bit mode of operation, the address size is always 64 bits, and while operating in 32-bit mode of operation, the address size is always 32 bits. Additionally, when operating in 16-bit addressing, memory operands used by enclave instructions use 32 bit addressing; the value of CS.D is ignored.

41.3.3 Interaction of Intel® SGX Instructions with Segmentation

All leaf functions of ENCLU and ENCLS instructions require that the DS segment be usable, and be an expand-up segment. Failing this check results in generation of a #GP(0) exception.

The Intel SGX leaf functions used for entering the enclave (ENCLU[EENTER] and ENCLU[ERESUME]) operate as follows:

- All usable segment registers except for FS and GS have a zero base.
- The contents of the FS/GS segment registers (including the hidden portion) is saved in the processor.
- New FS and GS values compatible with enclave security are loaded from the TCS
- The linear ranges and access rights available under the newly-loaded FS and GS must abide to OS policies by ensuring they are subsets of the linear-address range and access rights available for the DS segment.
- The CS segment mode (64-bit, compatible, or 32 bit modes) must be consistent with the segment mode for which the enclave was created, as indicated by the SECS.ATTRIBUTES.MODE64 bit, and that the CPL of the logical processor is 3

An exit from the enclave either via ENCLU[EEXIT] or via an AEX restores the saved values of FS/GS segment registers.

41.3.4 Interactions of Enclave Execution with Segmentation

During the course of execution, enclave code abides by all segmentation policies as dictated by IA32 and Intel 64 Architectures, and generates appropriate exceptions on violations.

Additionally, any attempt by software executing inside an enclave to modify the processor's segmentation state (e.g. via MOV seg register, POP seg register, LDS, far jump, etc; excluding WRFSBASE/WRGSBASE) results in the generation of a #UD. See Section 38.6.1 for more information.

Upon enclave entry via the EENTER leaf function, FS is loaded from the (TCS.OFSBASE + SECS.BASEADDR) and TCS.FSLIMIT fields and GS is loaded from the (TCS.OGSBASE + SECS.BASEADDR) and TCS.GSLIMIT fields.

Execution of WRFSBASE and WRGSBASE from inside a 64-bit enclave is allowed. The processor will save the new values into the current SSA frame on an asynchronous exit (AEX) and restore them back on enclave entry via ENCLU[ERESUME] instruction.

41.4 INTERACTIONS WITH PAGING

Intel SGX instructions are available only when the processor is executing in a protected mode of operation. Additionally, all Intel SGX leaf functions except for EDBGRD and EDBGWR are available only if paging is enabled. Any attempt to execute these leaf functions with paging disabled results in an invalid-opcode exception (#UD). As with

segmentation, enclaves abide by all the paging policies set up by the OS, but they can be more restrictive than the OS.

All the memory operands passed into Intel SGX instructions are interpreted as offsets within the DS segment, and the linear addresses generated by combining these offsets with DS segment register are subject to paging-based access control if paging is enabled at the time of the execution of the leaf function.

Since the ENCLU[EENTER] and ENCLU[ERESUME] can only be executed when paging is enabled, and since paging cannot be disabled by software running inside an enclave (recall that enclaves always run with CPL = 3), enclave execution is always subject to paging-based access control. The Intel SGX access control itself is implemented as an extension to the existing paging modes. See Section 37.5 for details.

Execution of Intel SGX instructions may set accessed and dirty flags on accesses to EPC pages that do not fault even if the instruction later causes a fault for some other reason.

41.5 INTERACTIONS WITH VMX

Intel SGX functionality (including SGX1 and SGX2) can be made available to software running in either VMX root operation or VMX non-root operation, as long as the processor is using a legal mode of operation (see Section 41.1).

A VMM has the flexibility to configure a VMCS to permit a guest to use any subset of the ENCLS leaf functions. Availability of the ENCLU leaf functions in VMX non-root operation has the same requirement as ENCLU leaf functions outside of a virtualized environment.

Details of the VMCS control to allow VMM to configure support of Intel SGX in VMX non-root operation is described in Section 41.5.1

41.5.1 VMM Controls to Configure Guest Support of Intel® SGX

Intel SGX capabilities are primarily exposed to the software via the CPUID instruction. VMMs can virtualize CPUID instruction to expose/hide this capability to/from guests.

Some of Intel SGX resources are exposed/controlled via model-specific registers (see Section 36.7). VMMs can virtualize these MSRs for the guests using the MSR bitmaps referenced by pointers in the VMCS.

The VMM can partition the Enclave Page Cache, and assign various partitions to (a subset of) its guests via the usual memory-virtualization techniques such as paging or the extended page table mechanism (EPT).

The VMM can set the "enable ENCLS exiting" VM-execution controls to cause a VM exit when the ENCLS instruction is executed in VMX non-root operation. If the "enable ENCLS exiting" control is 0, all of the ENCLS leaf functions are permitted in VMX non-root operation. If the "enable ENCLS exiting" control is 1, execution of ENCLS leaf functions in VMX non-root operation is governed by consulting the bits in a new 64-bit VM-execution control field called the ENCLS-exiting bitmap (Each bit in the bitmap corresponds to an ENCLS leaf function with an EAX value that is identical to the bit's position). When bits in the "ENCLS-exiting bitmap" are set, attempts to execute the corresponding ENCLS leaf functions in VMX non-root operation causes VM exits. The checking for these VM exits occurs immediately after checking that CPL = 0.

41.5.2 Interactions with the Extended Page Table Mechanism (EPT)

Intel SGX instructions are fully compatible with the extended page-table mechanism (EPT; see Section 28.2).

All the memory operands passed into Intel SGX instructions are interpreted as offsets within the DS segment, and the linear addresses generated by combining these offsets with DS segment register are subject to paging and EPT. As with paging, enclaves abide by all the policies set up by the VMM.

The Intel SGX access control itself is implemented as an extension to paging and EPT, and may be more restrictive. See Section 41.4 for details of this extension.

An execution of an Intel SGX instruction may set accessed and dirty flags for EPT (when enabled; see Section 28.2.4) on accesses to EPC pages that do not fault or cause VM exits even if the instruction later causes a fault or VM exit for some other reason.

41.5.3 Interactions with APIC Virtualization

This section applies to Intel SGX in VMX non-root operation when the "virtualize APIC accesses" VM-execution control is 1.

A memory access by an enclave instruction that implicitly uses a cached physical address is never checked for overlap with the APIC-access page. Such accesses never cause APIC-access VM exits and are never redirected to the virtual-APIC page. Implicit memory accesses can only be made to the SECS, the TCS, or the SSA of an enclave (see Section 37.5.3.2).

An explicit Enclave Access (a linear memory access which is either from within an enclave into its ELRANGE, or an access by an Intel SGX instruction that is expected to be in the EPC) that overlaps with the APIC-access page causes a #PF exception (APIC page is expected to be outside of EPC).

Non-Enclave accesses made either by an Intel SGX instruction or by a logical processor inside an enclave to an address that without SGX would have caused redirection to the virtual-APIC page instead cause an APIC-access VM exit.

Other than implicit accesses made by Intel SGX instructions, guest-physical and physical accesses are not considered "enclave accesses"; consequently, such accesses result in undefined behavior if these accesses eventually reach EPC. This applies to any non-enclave physical accesses.

While a logical processor is executing inside an enclave, an attempt to execute an instruction outside of ELRANGE results in a #GP(0), even if the linear address would translate to a physical address that overlaps the APIC-access page.

41.5.4 Interactions with VT and SGX concurrency

In some cases, a VMM is required to handle conflicts between its own operation and a guest operation on EPC pages that are present in both guest and VMM address space. These conflict would otherwise cause the guest to experience an unexpected behavior (vs. running directly on the h/w). These conflict cases are:

- ETRACK/ETRACKC failure due to Entry Epoch Object Lock conflict or reference tracking check failure.
- EPC Page Resource conflict.

A new exit reason is defined for all those cases: SGX_CONFLICT (value 71). The VMCS exit qualification field details the specific case as follows:

Table 41-1. SGX Conflict Exit Qualification

Bits	Size (bits)	Name	Description
15:0	16	Code	Exit qualification code. The following values are defined: 0: TRACKING_RESOURCE_CONFLICT 1: TRACKING_REFERENCE_CONFLICT 2: EPC_PAGE_CONFLICT_EXCEPTION 3: EPC_PAGE_CONFLICT_ERROR Other: Reserved
31:16	16	Error	Error code. Applicable only if the exit qualification code is EPC_PAGE_CONFLICT_ERROR; contains the error code that would be returned in RAX if the instruction was executed on bare metal platform or if the ENABLE_EPC_VIRTUALIZATION_EXTENSIONS bit in the secondary processor control field is not set. In other cases this field is reserved as 0.
63:32	32	Reserved	Always 0.

This SGX_CONFLICT exiting behavior is controlled by a VM execution control called ENABLE_EPC_VIRTUALIZATION_EXTENSIONS (bit 29 of the secondary processor control field).

Details for various SGX_CONFLICT VMEXIT cases are provided in the following sections.

41.5.5 Virtual Child Tracking

SGX oversubscription support adds the ability to associate virtual children with each enclave using the ENCLV[EINCVIRTCHILD] and ENCLV[EDECVIRTCHILD] instructions. The VMM enables checking of the virtual child count by EREMOVE and EWB in guests with a new VM execution control called ENABLE_EPC_VIRTUALIZATION_EXTENSIONS.

When in VMX non-root operation and the ENABLE_EPC_VIRTUALIZATION_EXTENSIONS control enabled, the following instructions change their behavior:

- EWB and EREMOVE return the SGX_CHILD_PRESENT error code if any virtual or physical children are associated with the enclave.
- ERDINFO set STATUS.CHILDPRESENT if any virtual or physical children are associated with the enclave.

41.5.6 Handling EPCM Entry Lock Conflicts

When performing paging within a VMM, it is possible for a contention on the EPC page to happen in the following case:

- The VMM performs an ELDB/ELDU/ELDBC/ELDUC of an enclave page, and the guest attempts to perform some SGX instruction (e.g., EREMOVE) where the same SECS parent page is required.

A similar conflict may occur if the VMM uses EINCVIRTCHILD or EDECVIRTCHILD pointing to an SECS page. In all other cases where a SGX instruction executed by the VMM the applicable EPC page should not be mapped to the guest, thus no resource conflict occurs.

This conflicting situation can cause the guest's instruction to fail and cause guest instability. To help the VMM manage such conflicts, the SGX VMM paging extensions introduce a new VM-Exit that will be triggered whenever the guest encounters a resource conflict.

The exit reason is SGX_CONFLICT. The exit qualification field is used to distinguish the two kinds of resource conflicts:

- A value of EPCM_RESOURCE_CONFLICT_EXCEPTION (2) in the exit qualification code field indicates that a resource conflict occurred that would result in a #GP. In that case, the exit qualification error field is set to zero.
- A value of EPC_PAGE_CONFLICT_ERROR (3) in the exit qualification code field indicates that a resource conflict occurred that would result in an error code being return in RAX. In that case, the exit qualification error field is set to SGX_EPC_PAGE_CONFLICT.

The Guest Linear Address and Guest Physical Address fields are set to the guest linear and guest physical addresses respectively of the EPC page on which the conflict occurred. The VMM may determine which instruction induced the exit by reading RAX. The exit also populates the VM-exit instruction length field.

The VMM can determine whether the conflict may be due to its own operation, e.g., by setting a per-enclave busy indicator before executing ELD*, and clearing it afterwards. In that case, the VMM can handle an SGX Conflict (EPCM_PAGE_CONFLICT_*) exit by resuming guest execution at the same instruction, allowing the guest to re-execute the instruction. The VMM may also take steps to throttle its own paging thread to reduce contention with the guest.

If the VMM determines that the conflict is not due to its own operation, it may inject a #GP (in case of EPC_PAGE_CONFLICT_EXCEPTION), or emulate an error code as the guest instruction would return (in case of EPC_PAGE_CONFLICT_ERROR) by setting ZF and copying the error value provided in the exit qualification to guest RAX.

To gracefully handle resource contention on the VMM side, the VMM should use the new ELDBC and ELDUC instructions. These are similar to ELDB and ELDU respectively, except that on EPC resource contention they return an SGX_EPC_PAGE_CONFLICT error instead of issuing a #GP. In case of an error, the VMM can retry the instruction, possibly throttling the guest to assure progress.

When using EDECVIRTCHILD and EINCVIRTCHILD, the VMM should preferably point to the enclave child page, not to the SECS page, avoiding resource conflict on the SECS. If the VMM chooses to point to the SECS page, it should handle conflicts in the same way as handling the ELD* case.

41.5.7 Context Tracking

The ENCLAVECONTEXT field in the SECS is available for use by the VMM to track context information associated with that enclave, such as the GPA of the SECS in the context of the appropriate guest. This field is initialized by the successful execution of ECREATE and ELD of an SECS page. The value stored in the ENCLAVECONTEXT field will be the translation of the target page address produced by paging (GPA in VMMs that have EPTs turned on). VMMs may override this default value by calling the ENCLV[ESETCONTEXT] instruction, which allows the VMM to store an arbitrary 64-bit value in the ENCLAVECONTEXT field. The VMM may later access the ENCLAVECONTEXT field by calling ENCLS[ERDINFO] on any member page of the enclave, including the SECS.

For nested virtualization cases, the lowest level VMM can make SGX oversubscription instructions higher level guest VMMs. In that case the lower level VMM can simply inject #GP to higher level VMMs when attempting to execute these instructions.

However, if VMMs expose SGX oversubscription instructions to higher level VMMs, then VMMs have to use ENCLV[ESETCONTEXT] instruction to properly manage the ENCLAVECONTEXT field of SECS during paging operations. That may involve emulating ERCREATE, ELD, ESETCONTEXT and ERDINFO instructions apart from managing ENCLAVECONTEXT values.

41.6 INTEL® SGX INTERACTIONS WITH ARCHITECTURALLY-VISIBLE EVENTS

All architecturally visible vectored events (IA32 exceptions, interrupts, SMI, NMI, INIT, VM exit) can be detected while inside an enclave and will cause an asynchronous enclave exit if they are not blocked. Additionally, INT3, and the SignalTXTMsg[SENTER] (i.e. GETSEC[SENTER]'s rendezvous event message) events also cause asynchronous enclave exits. Note that SignalTXTMsg[SEXIT] (i.e. GETSEC[SEXIT]'s teardown message) does not cause an AEX.

On an AEX, information about the event causing the AEX is stored in the SSA (see Section 39.4 for details of AEX). The information stored in the SSA only describes the first event that triggered the AEX. If parsing/delivery of the first event results in detection of further events (e.g. VM exit, double fault, etc.), then the event information in the SSA is not updated to reflect these subsequently detected events.

41.7 INTERACTIONS WITH THE PROCESSOR EXTENDED STATE AND MISCELLANEOUS STATE

41.7.1 Requirements and Architecture Overview

Processor extended states are the ISA features that are enabled by the settings of CR4.OSXSAVE and the XCR0 register. Processor extended states are normally saved/restored by software via XSAVE/XRSTOR instructions. Details of discovery of processor extended states and management of these states are described in CHAPTER 13 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Additionally, the following requirements apply to Intel SGX:

- On an AEX, the Intel SGX architecture must protect the processor extended state and miscellaneous state by saving them in the enclave's state-save area (SSA), and clear the secrets from the processor extended state that is used by an enclave.
- Intel SGX architecture must verify that the SSA frame size is large enough to contain all the processor extended states and miscellaneous state used by the enclave.
- Intel SGX architecture must ensure that enclaves can only use processor extended state that is enabled by system software in XCR0.
- Enclave software should be able to discover only those processor extended state and miscellaneous state for which such protection is enabled.
- The processor extended states that are enabled inside the enclave must be approved by the enclave developer:
  - Certain processor extended state (e.g., Memory Protection Extensions, see Chapter 17, "Intel® MPX" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) modify the behavior of the

legacy ISA software. If such features are enabled for enclaves that do not understand those features, then such a configuration could lead to a compromise of the enclave's security.

- The processor extended states that are enabled inside the enclave must form an integral part of the enclave's identity. This requirement has two implications:
  - Service providers may decide to assign different trust level to the same enclave depending on the ISA features the enclave is using.

To meet these requirements, the Intel SGX architecture defines a sub-field called X-Feature Request Mask (XFRM) in the ATTRIBUTES field of the SECS. On enclave creation (ENCLS[ECREATE] leaf function), the required SSA frame size is calculated by the processor from the list of enabled extended and miscellaneous states and verified against the actual SSA frame size defined by SECS.SSAFRAMESIZE.

On enclave entry, after verifying that XFRM is only enabling features that are already enabled in XCR0, the value in the XCR0 is saved internally by the processor, and is replaced by the XFRM. On enclave exit, the original value of XCR0 is restored. Consequently, while inside the enclave, the processor extended states enabled in XFRM are in enabled state, and those that are disabled in XFRM are in disabled state.

The entire ATTRIBUTES field, including the XFRM subfield is integral part of enclave's identity (i.e., its value is included in reports generated by ENCLU[EREPORT], and select bits from this field can be included in key-derivation for keys obtained via the ENCLU[EGETKEY] leaf function).

Enclave developers can create their enclave to work with certain features and fallback to another code path in case those features aren't available (e.g. optimize for AVX and fallback to SSE). For this purpose Intel SGX provides the following fields in SIGSTRUCT: ATTRIBUTES, ATTRIBUTESMASK, MISCSELECT, and MISCMASK. EINIT ensures that the final SECS.ATTRIBUTES and SECS.MISCSELECT comply with the enclave developer's requirements as follows:

SIGSTRUCT.ATTRIBUTES & SIGSTRUCT.ATTRIBUTEMASK = SECS.ATTRIBUTES & SIGSTRUCT.ATTRIBUTEMASK

SIGSTRUCT.MISCSELECT & SIGSTRUCT.MISCMASK = SECS.MISCSELECT & SIGSTRUCT.MISCMASK.

On an asynchronous enclave exit, the processor extended states enabled by XFRM are saved in the current SSA frame, and overwritten by synthetic state (see Section 39.3 for the definition of the synthetic state). When the interrupted enclave is resumed via the ENCLU[ERESUME] leaf function, the saved state for processor extended states enabled by XFRM is restored.

41.7.2 Relevant Fields in Various Data Structures

41.7.2.1 SECS.ATTRIBUTES.XFRM

The ATTRIBUTES field of the SECS data structure (see Section 37.7) contains a sub-field called XSAVE-Feature Request Mask (XFRM). Software populates this field at the time of enclave creation according to the features that are enabled by the operating system and approved by the enclave developer.

Intel SGX architecture guarantees that during enclave execution, the processor extended state configuration of the processor is identical to what is required by the XFRM sub-field. All the processor extended states enabled in XFRM are saved on AEX from the enclave and restored on ERESUME.

The XFRM sub-field has the same layout as XCR0, and has consistency requirements that are similar to those for XCR0. Specifically, the consistency requirements on XFRM values depend on the processor implementation and the set of features enabled in CR4.

Legal values for SECS.ATTRIBUTES.XFRM conform to these requirements:

- XFRM[1:0] must be set to 0x3.
- If the processor does not support XSAVE, or if the system software has not enabled XSAVE, then XFRM[63:2] must be zero.
- If the processor does support XSAVE, XFRM must contain a value that would be legal if loaded into XCR0.

The various consistency requirements are enforced at different times in the enclave's life cycle, and the exact enforcement mechanisms are elaborated in Section 41.7.3 through Section 41.7.6.

On processors not supporting XSAVE, software should initialize XFRM to 0x3. On processors supporting XSAVE, software should initialize XFRM to be a subset of XCR0 that would be present at the time of enclave execution.

Because bits 0 and 1 of XFRM must always be set, the use of Intel SGX requires that SSE be enabled (CR4.OSFXSR = 1).

41.7.2.2 SECS.SSAFRAMESIZE

The SSAFRAMESIZE field in the SECS data structure specifies the number of pages which software allocated[1] for each SSA frame, including both the GPRSGX area, MISC area, the XSAVE area (x87 and XMM states are stored in the latter area), and optionally padding between the MISC and XSAVE area. The GPRSGX area must hold all the general-purpose registers and additional Intel SGX specific information. The MISC area must hold the Miscellaneous state as specified by SECS.MISCSELECT, the XSAVE area holds the set of processor extended states specified by SECS.ATTRIBUTES.XFRM (see Section 37.9 for the layout of SSA and Section 41.7.3 for ECREATE's consistency checks). The SSA is always in non-compacted format.

If the processor does not support XSAVE, the XSAVE area will always be 576 bytes; a copy of XFRM (which will be set to 0x3) is saved at offset 512 on an AEX.

If the processor does support XSAVE, the length of the XSAVE area depends on SECS.ATTRIBUTES.XFRM. The length would be equal to what CPUID.(EAX=0DH, ECX= 0):EBX would return if XCR0 were set to XFRM. The following pseudo code illustrates how software can calculate this length using XFRM as the input parameter without modifying XCR0:

```
offset = 576;
size_last_x = 0;
For x=2 to 63
IF (XFRM[x] != 0) Then
   tmp_offset = CPUID.(EAX=0DH, ECX= x):EBX[31:0];
    IF (tmp_offset >= offset + size_last_x) Then
        offset = tmp_offset;
        size_last_x = CPUID.(EAX=0DH, ECX= x):EAX[31:0];
     FI;
FI;
EndFor
return (offset + size_last_x); (* compute_xsave_size(XFRM), see "ECREATE—Create an SECS page in the Enclave Page Cache"*)
```

Where the non-zero bits in XFRM are a subset of non-zero bit fields in XCR0.

The size of the MISC region depends on the setting of SECS.MISCSELECT and can be calculated using the layout information described in Section 37.9.2

41.7.2.3 XSAVE Area in SSA

The XSAVE area of an SSA frame begins at offset 0 of the frame.

41.7.2.4 MISC Area in SSA

The MISC area of an SSA frame is positioned immediately before the GPRSGX region.

41.7.2.5 SIGSTRUCT Fields

Intel SGX provides the flexibility for an enclave developer to choose the enclave's code path according to the features that are enabled on the platform (e.g. optimize for AVX and fallback to SSE). See Section 41.7.1 for details.

1. It is the responsibility of the enclave to actually allocate this memory.

SIGSTRUCT includes the following fields:

SIGSTRUCT.ATTRIBUTES, SIGSTRUCT.ATTRIBUTEMASK, SIGSTRUCT.MISCSELECT, SIGSTRUCT.MISCMASK.

41.7.2.6 REPORT.ATTRIBUTES.XFRM and REPORT.MISCSELECT

The processor extended states and miscellaneous states that are enabled inside the enclave form an integral part of the enclave's identity and are therefore included in the enclave's report, as provided by the ENCLU[EREPORT] leaf function. The REPORT structure includes the enclave's XFRM and MISCSELECT configurations.

41.7.2.7 KEYREQUEST

An enclave developer can specify which bits out of XFRM and MISCSELECT ENCLU[EGETKEY] should include in the derivation of the sealing key by specifying ATTRIBUTESMASK and MISCMASK in the KEYREQUEST structure.

41.7.3 Processor Extended States and ENCLS[ECREATE]

The ECREATE leaf function of the ENCLS instruction enforces a number of consistency checks described earlier. The execution of ENCLS[ECREATE] leaf function results in a #GP(0) in any of the following cases:

- SECS.ATTRIBUTES.XFRM[1:0] is not 3.
- The processor does not support XSAVE and any of the following is true:
  - SECS.ATTRIBUTES.XFRM[63:2] is not 0.
  - SECS.SSAFRAMESIZE is 0.
- The processor supports XSAVE and any of the following is true:
  - XSETBV would fault on an attempt to load XFRM into XCR0.
  - XFRM[63]=1.
  - The SSAFRAME is too small to hold required, enabled states (see Section 41.7.2.2).

41.7.4 Processor Extended States and ENCLU[EENTER]

41.7.4.1 Fault Checking

The EENTER leaf function of the ENCLU instruction enforces a number of consistency requirements described earlier. The execution of the ENCLU[EENTER] leaf function results in a #GP(0) in any of the following cases:

- If CR4.OSFXSR=0.
- If The processor supports XSAVE and either of the following is true:
  - CR4.OSXSAVE=0 and SECS.ATTRIBUTES.XFRM is not 3.
  - (SECS.ATTRIBUTES.XFRM & XCR0) != SECS.ATTRIBUTES.XFRM

41.7.4.2 State Loading

If ENCLU[EENTER] is successful, the current value of XCR0 is saved internally by the processor and replaced by SECS.ATTRIBUTES.XFRM.

41.7.5 Processor Extended States and AEX

41.7.5.1 State Saving

On an AEX, processor extended states are saved into the XSAVE area of the SSA frame in a compatible format with XSAVE that was executed with EDX:EAX = SECS.ATTRIBUTES.XFRM, with the memory operand being the XSAVE area, and (for 64-bit enclaves) as if REX.W=1. The XSTATE_BV part of the XSAVE header is saved with 0 for every bit that is 0 in XFRM. Other bits may be saved as 0 if the state saved is initialized.

Note that enclave entry ensures that if CR4.OSXSAVE is set to 0, then SECS.ATTRIBUTES.XFRM is set to 3. It should also be noted that it is not possible to enter an enclave with FXSAVE disabled.

41.7.5.2 State Synthesis

After saving the extended state, the processor restores XCR0 to the value it held at the time of the most recent enclave entry.

The state of features corresponding to bits set in XFRM is synthesized. In general, these states are initialized. Details of state synthesis on AEX are documented in Section 39.3.1.

41.7.6 Processor Extended States and ENCLU[ERESUME]

41.7.6.1 Fault Checking

The ERESUME leaf function of the ENCLU instruction enforces a number of consistency requirements described earlier. Specifically, the ENCLU[ERESUME] leaf function results in a #GP(0) in any of the following cases:

- CR4.OSFXSR=0.
- The processor supports XSAVE and either of the following is true:
  - CR4.OSXSAVE=0 and SECS.ATTRIBUTES.XFRM is not 3.
  - (SECS.ATTRIBUTES.XFRM & XCR0) != SECS.ATTRIBUTES.XFRM.

A successful execution of ENCLU[ERESUME] loads state from the XSAVE area of the SSA frame in a fashion similar to that used by the XRSTOR instruction. Data in the XSAVE area that would cause the XRSTOR instruction to fault will cause the ENCLU[ERESUME] leaf function to fault. Examples include, but are not restricted to the following:

- A bit is set in the XSTATE_BV field and clear in XFRM.
- The required bytes in the header are not clear.
- Loading data would set a reserved bit in MXCSR.

Any of these conditions will cause ERESUME to fault, even if CR4.OSXSAVE=0.

41.7.6.2 State Loading

If ENCLU[ERESUME] is successful, the current value of XCR0 is saved internally by the processor and replaced by SECS.ATTRIBUTES.XFRM.

State is loaded from the XSAVE area of the SSA frame as if the XRSTOR instruction were executed with XCR0=XFRM, EDX:EAX = XFRM, with the memory operand being the XSAVE area, and (for 64-bit enclaves) as if REX.W=1.

ENCLU[ERESUME] ensures that a subsequent execution of XSAVEOPT inside the enclave will operate properly (e.g., by marking all state as modified).

41.7.7 Processor Extended States and ENCLU[EEXIT]

The ENCLU[EEXIT] leaf function does not perform any X-feature specific consistency checks, nor performs any state synthesis. It is the responsibility of enclave software to clear any sensitive data from the registers before

executing EEXIT. However, successful execution of the ENCLU[EEXIT] leaf function restores XCR0 to the value it held at the time of the most recent enclave entry.

41.7.8 Processor Extended States and ENCLU[EREPORT]

The ENCLU[EREPORT] leaf function creates the MAC-protected REPORT structure that reports on the enclave's identity. ENCLU[EREPORT] includes in the report the values of SECS.ATTRIBUTES.XFRM and SECS.MISCSELECT.

41.7.9 Processor Extended States and ENCLU[EGETKEY]

The ENCLU[EGETKEY] leaf function returns a cryptographic key based on the information provided by the KEYREQUEST structure. Intel SGX provides the means for isolation between different operating conditions by allowing an enclave developer to select which bits out of XFRM and MISCSELECT need to be included in the derivation of the keys.

41.8 INTERACTIONS WITH SMM

41.8.1 Availability of Intel® SGX instructions in SMM

Enclave instructions are not available in SMM, and any attempt to execute ENCLS or ENCLU instructions inside SMM results in an invalid-opcode exception (#UD).

41.8.2 SMI while Inside an Enclave

If the logical processor executing inside an enclave receives an SMI, the logical processor exits the enclave asynchronously. The response to an SMI received while executing inside an enclave depends on whether the dual-monitor treatment is enabled. For detailed discussion of transfer to SMM, see Chapter 34, "System Management Mode" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

If the logical processor executing inside an enclave receives an SMI when dual-monitor treatment is not enabled, the logical processor exits the enclave asynchronously, and transfers the control to the SMM handler. In addition to saving the synthetic architectural state to the SMRAM State Save Map (SSM), the logical processor also sets the "Enclave Interruption" bit in the SMRAM SSM (bit position 1 in SMRAM field at offset 7EE0H).

If the logical processor executing inside an enclave receives an SMI when dual-monitor treatment is enabled, the logical processor exits the enclave asynchronously, and transfers the control to the SMM monitor via SMM VM exit. The SMM VM exit sets the "Enclave Interruption" bit in the Exit Reason (see Table 41-2) and in the Guest Interruptibility State field (see Table 41-3) of the SMM VMCS.

41.8.3 SMRAM Synthetic State of AEX Triggered by SMI

All processor registers saved in the SMRAM have the same synthetic values listed in Section 39.3. Additional SMRAM fields that are treated specially on SMI are:

Table 41-2. SMRAM Synthetic States on Asynchronous Enclave Exit

Position	Field	Value	Writable
SMRAM Offset 07EE0H.Bit 1	ENCLAVE_INTERRUPTION	Set to 1 if exit occurred in enclave mode	No

41.9 INTERACTIONS OF INIT, SIPI, AND WAIT-FOR-SIPI WITH INTEL® SGX

INIT received inside an enclave, while the logical processor is not in VMX operation, causes the logical processor to exit the enclave asynchronously. After the AEX, the processor's architectural state is initialized to "Power-on" state (Table 9.1 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). If the logical processor is BSP, then it proceeds to execute the BIOS initialization code. If the logical processor is an AP, it enters wait-for-SIPI state.

INIT received inside an enclave, while the logical processor (LP) is in VMX root operation, follows regular Intel Architecture behavior and is blocked.

INIT received inside an enclave, while the logical processor is in VMX non-root operation, causes an AEX. Subsequent to the AEX, the INIT causes a VM exit with the Enclave Interruption bit in the exit-reason field in the VMCS.

A processor cannot be inside an enclave in the wait-for-SIPI state. Consequently, a SIPI received while inside an enclave is lost.

Intel SGX does not change the behavior of the processor in the wait-for-SIPI state.

The SGX-related processor states after INIT-SIPI-SIPI is as follows:

- EPC Settings: Unchanged
- EPCM: Unchanged
- CPUID.LEAF_12H.*: Unchanged
- ENCLAVE_MODE: 0 (LP exits enclave asynchronously)
- MEE state: Unchanged

Software should be aware that following INIT-SIPI-SIPI, the EPC might contain valid pages and should take appropriate measures such as initialize the EPC with the EREMOVE leaf function.

41.10 INTERACTIONS WITH DMA

DMA is not allowed to access any Processor Reserved Memory.

41.11 INTERACTIONS WITH TXT

41.11.1 Enclaves Created Prior to Execution of GETSEC

Enclaves which have been created before the GETSEC[SENTER] leaf function are available for execution after the successful completion of GETSEC[SENTER] and the corresponding SINIT ACM. Actions that a TXT Launched Environment performs in preparation to execute code in the Launched Environment, also applies to enclave code that would run after GETSEC[SENTER].

41.11.2 Interaction of GETSEC with Intel® SGX

All leaf functions of the GETSEC instruction are illegal inside an enclave, and results in an invalid-opcode exception (#UD).

Responding Logical Processors (RLP) which are executing inside an enclave at the time a GETSEC[SENTER] event occurs perform an AEX from the enclave and then enter the Wait-for-SIPI state.

RLP executing inside an enclave at the time of GETSEC[SEXIT], behave as defined for GETSEC[SEXIT]-that is, the RLPs pause during execution of SEXIT and resume after the completion of SEXIT.

The execution of a TXT launch does not affect Intel SGX configuration or security parameters.

41.11.3 Interactions with Authenticated Code Modules (ACMs)

Intel SGX only allows launching ACMs with an Intel SGX SVN that is at the same level or higher than the expected Intel SGX SVN. The expected Intel SGX SVN is specified by BIOS and locked down by the processor on the first successful execution of an Intel SGX instruction that doesn't return an error code. Intel SGX provides interfaces for system software to discover whether a non faulting Intel SGX instruction has been executed, and evaluate the suitability of the Intel SGX SVN value of any ACM that is expected to be launched by the OS or the VMM.

These interfaces are provided through a read-only MSR called the IA32_SGX_SVN_STATUS MSR (MSR address 500h). The IA32_SGX_SVN_STATUS MSR has the format shown in Table 41-3.

Table 41-3. Layout of the IA32_SGX_SVN_STATUS MSR

Bit Position	Name	ACM Module ID	Value
0	Lock	N.A.	▪ If 1, indicates that a non-faulting Intel SGX instruction has been executed, consequently, launching a properly signed ACM but with Intel SGX SVN value less than the BIOS specified Intel SGX SVN threshold would lead to an TXT shutdown. ▪ If 0, indicates that the processor will allow a properly signed ACM to launch irrespective of the Intel SGX SVN value of the ACM.
15:1	RSVD	N.A.	0
23:16	SGX_SVN_SINIT	SINIT ACM	▪ If CPUID.01H:ECX.SMX =1, this field reflects the expected threshold of Intel SGX SVN for the SINIT ACM. ▪ If CPUID.01H:ECX.SMX =0, this field is reserved (0).
63:24	RSVD	N.A.	0

OS/VMM that wishes to launch an architectural ACM such as SINIT is expected to read the IA32_SGX_SVN_STATUS MSR to determine whether the ACM can be launched or a new ACM is needed:

- If either the Intel SGX SVN of the ACM is greater than the value reported by IA32_SGX_SVN_STATUS, or the lock bit in the IA32_SGX_SVN_STATUS is not set, then the OS/VMM can safely launch the ACM.
- If the Intel SGX SVN value reported in the corresponding component of the IA32_SGX_SVN_STATUS is greater than the Intel SGX SVN value in the ACM's header, and if bit 0 of IA32_SGX_SVN_STATUS is 1, then the OS/VMM should not launch that version of the ACM. It should obtain an updated version of the ACM either from the BIOS or from an external resource.

However, OSVs/VMMs are strongly advised to update their version of the ACM any time they detect that the Intel SGX SVN of the ACM carried by the OS/VMM is lower than that reported by IA32_SGX_SVN_STATUS MSR, irrespective of the setting of the lock bit.

41.12 INTERACTIONS WITH CACHING OF LINEAR-ADDRESS TRANSLATIONS

Entering and exiting an enclave causes the logical processor to flush all the global linear-address context as well as the linear-address context associated with the current VPID and PCID. The MONITOR FSM is also cleared.

41.13 INTERACTIONS WITH INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

1. ENCLU or ENCLS instructions inside an HLE region will cause the flow to be aborted and restarted non-speculatively. ENCLU or ENCLS instructions inside an RTM region will cause the flow to be aborted and transfer control to the fallback handler.

2. If XBEGIN is executed inside an enclave, the processor does NOT check whether the address of the fallback handler is within the enclave.

3. If an RTM transaction is executing inside an enclave and there is an attempt to fetch an instruction outside the enclave, the transaction is aborted and control is transferred to the fallback handler. No #GP is delivered.

4. If an RTM transaction is executing inside an enclave and there is a data access to an address within the enclave that denied due to EPCM content (e.g., to a page belonging to a different enclave), the transaction is aborted and control is transferred to the fallback handler. No #GP is delivered.

5. If an RTM transaction executing inside an enclave aborts and the address of the fallback handler is outside the enclave, a #GP is delivered after the abort (EIP reported is that of the fallback handler).

41.13.1 HLE and RTM Debug

RTM debug will be suppressed on opt-out enclave entry. After opt-out entry, the logical processor will behave as if IA32_DEBUG_CTL[15]=0. Any #DB detected inside an RTM transaction region will just cause an abort with no exception delivered.

After opt-in entry, if either DR7[11] = 0 OR IA32_DEBUGCTL[15] = 0, any #DB or #BP detected inside an RTM transaction region will just cause an abort with no exception delivered.

After opt-in entry, if DR7[11] = 1 AND IA32_DEBUGCTL[15] = 1, any #DB or #BP detected inside an RTM translation will

- terminate speculative execution,
- set RIP to the address of the XBEGIN instruction, and
- be delivered as #DB (implying an Intel SGX AEX; any #BP is converted to #DB).
- DR6[16] will be cleared, indicating RTM debug (if the #DB causes a VM exit, DR6 is not modified but bit 16 of the pending debug exceptions field in the VMCS will be set).

41.14 INTEL® SGX INTERACTIONS WITH S STATES

Whenever an Intel SGX enabled processor enters S3-S5 state, enclaves are destroyed. This is due to the EPC being destroyed when power down occurs. It is the application runtime's responsibility to re-instantiate an enclave after a power transition for which the enclaves were destroyed.

41.15 INTEL® SGX INTERACTIONS WITH MACHINE CHECK ARCHITECTURE (MCA)

41.15.1 Interactions with MCA Events

All architecturally visible machine check events (#MC and CMCI) that are detected while inside an enclave cause an asynchronous enclave exit.

Any machine check exception (#MC) that occurs after Intel SGX is first enables causes Intel SGX to be disabled, (CPUID.SGX_Leaf.0:EAX[SGX1] == 0). It cannot be enabled until after the next reset.

41.15.2 Machine Check Enables (IA32_MCi_CTL)

All supported IA32_MCi_CTL bits for all the machine check banks must be set for Intel SGX to be available (CPUID.SGX_Leaf.0:EAX[SGX1] == 1). Any act of clearing bits from '1 to '0 in any of the IA32_MCi_CTL register may disable Intel SGX (set CPUID.SGX_Leaf.0:EAX[SGX1] to 0) until the next reset.

41.15.3 CR4.MCE

CR4.MCE can be set or cleared with no interactions with Intel SGX.

41.16 INTEL® SGX INTERACTIONS WITH PROTECTED MODE VIRTUAL INTERRUPTS

ENCLS[EENTER] modifies neither EFLAGS.VIP nor EFLAGS.VIF.

ENCLS[ERESUME] loads EFLAGS in a manner similar to that of an execution of IRET with CPL = 3. This means that ERESUME modifies neither EFLAGS.VIP nor EFLAGS.VIF regardless of the value of the EFLAGS image in the SSA frame.

AEX saves EFLAGS.VIP and EFLAGS.VIF unmodified into the EFLAGS image in the SSA frame. AEX modifies neither EFLAGS.VIP nor EFLAGS.VIF after saving EFLAGS.

If CR4.PVI = 1, CPL = 3, EFLAGS.VM = 0, IOPL < 3, EFLAGS.VIP = 1, and EFLAGS.VIF = 0, execution of STI causes a #GP fault. In this case, STI modifies neither EFLAGS.IF nor EFLAGS.VIF. This behavior applies without change within an enclave (where CPL is always 3). Note that, if IOPL = 3, STI always sets EFLAGS.IF without fault; CR4.PVI, EFLAGS.VIP, and EFLAGS.VIF are neither consulted nor modified in this case.

41.17 INTEL SGX INTERACTION WITH PROTECTION KEYS

SGX interactions with PKRU are as follows:

- CPUID.(EAX=12H, ECX=1):ECX.PKRU indicates whether SECS.ATTRIBUTES.XFRM.PKRU can be set. If SECS.ATTRIBUTES.XFRM.PKRU is set, then PKRU is saved and cleared as part of AEX and is restored as part of ERESUME. If CR4.PKE is set, an enclave can execute RDPKRU and WRKRU independent of whether SECS.ATTRIBUTES.XFRM.PKRU is set.

SGX interactions with domain permission checks are as follows:

1) If CR4.PKE is not set, then legacy and SGX permission checks are not effected.

2) If CR4.PKE is set, then domain permission checks are applied to all non-enclave access and enclave accesses to user pages in addition to legacy and SGX permission checks at a higher priority than SGX permission checks.

3) Implicit accesses aren't subject to domain permission checks.

28. Updates to Appendix A, Volume 3D

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Change to this chapter: Deletion of duplicate paragraph.

APPENDIX A
VMX CAPABILITY REPORTING FACILITY

The ability of a processor to support VMX operation and related instructions is indicated by CPUID.1:ECX.VMX[bit 5] = 1. A value 1 in this bit indicates support for VMX features.

Support for specific features detailed in Chapter 26 and other VMX chapters is determined by reading values from a set of capability MSRs. These MSRs are indexed starting at MSR address 480H. VMX capability MSRs are read-only; an attempt to write them (with WRMSR) produces a general-protection exception (#GP(0)). They do not exist on processors that do not support VMX operation; an attempt to read them (with RDMSR) on such processors produces a general-protection exception (#GP(0)).

A.1 BASIC VMX INFORMATION

The IA32_VMX_BASIC MSR (index 480H) consists of the following fields:

- Bits 30:0 contain the 31-bit VMCS revision identifier used by the processor. Processors that use the same VMCS revision identifier use the same size for VMCS regions (see subsequent item on bits 44:32).[1]
- Bit 31 is always 0.
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear).
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width.[2] If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture.
- If bit 49 is read as 1, the logical processor supports the dual-monitor treatment of system-management interrupts and system-management mode. See Section 34.15 for details of this treatment.
- Bits 53:50 report the memory type that should be used for the VMCS, for data structures referenced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and for the MSEG header. If software needs to access these data structures (e.g., to modify the contents of the MSR bitmaps), it can configure the paging structures to map them into the linear-address space. If it does so, it should establish mappings that use the memory type reported bits 53:50 in this MSR.[3]

  As of this writing, all processors that support VMX operation indicate the write-back type. The values used are given in Table A-1.

Table A-1. Memory Types Recommended for VMCS and Related Data Structures

Value(s)	Field
0	Uncacheable (UC)
1–5	Not used
6	Write Back (WB)
7–15	Not used

1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field in bits 31:0 of this MSR. For all processors produced prior to this change, bit 31 of this MSR was read as 0.
2. On processors that support Intel 64 architecture, the pointer must not set bits beyond the processor's physical address width.
3. Alternatively, software may map any of these regions or structures with the UC memory type. (This may be necessary for the MSEG header.) Doing so is discouraged unless necessary as it will cause the performance of software accesses to those structures to suffer.

- If bit 54 is read as 1, the processor reports information in the VM-exit instruction-information field on VM exits due to execution of the INS and OUTS instructions (see Section 27.2.4). This reporting is done only if this bit is read as 1.
- Bit 55 is read as 1 if any VMX controls that default to 1 may be cleared to 0. See Appendix A.2 for details. It also reports support for the VMX capability MSRs IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3.1, Appendix A.3.2, Appendix A.4, and Appendix A.5 for details.
- The values of bits 47:45 and bits 63:56 are reserved and are read as 0.

A.2 RESERVED CONTROLS AND DEFAULT SETTINGS

As noted in Chapter 26, "VM Entries", certain VMX controls are reserved and must be set to a specific value (0 or 1) determined by the processor. The specific value to which a reserved control must be set is its **default setting**. Software can discover the default setting of a reserved control by consulting the appropriate VMX capability MSR (see Appendix A.3 through Appendix A.5).

Future processors may define new functionality for one or more reserved controls. Such processors would allow each newly defined control to be set either to 0 or to 1. Software that does not desire a control's new functionality should set the control to its default setting. For that reason, it is useful for software to know the default settings of the reserved controls.

Default settings partition the various controls into the following classes:

- **Always-flexible**. These have never been reserved.
- **Default0**. These are (or have been) reserved with a default setting of 0.
- **Default1**. They are (or have been) reserved with a default setting of 1.

As noted in Appendix A.1, a logical processor uses bit 55 of the IA32_VMX_BASIC MSR to indicate whether any of the default1 controls may be 0:

- If bit 55 of the IA32_VMX_BASIC MSR is read as 0, all the default1 controls are reserved and must be 1. VM entry will fail if any of these controls are 0 (see Section 26.2.1).
- If bit 55 of the IA32_VMX_BASIC MSR is read as 1, not all the default1 controls are reserved, and some (but not necessarily all) may be 0. The CPU supports four (4) new VMX capability MSRs: IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3 through Appendix A.5 for details. (These MSRs are not supported if bit 55 of the IA32_VMX_BASIC MSR is read as 0.)

See Section 31.5.1 for recommended software algorithms for proper capability detection of the default1 controls.

A.3 VM-EXECUTION CONTROLS

There are separate capability MSRs for the pin-based VM-execution controls, the primary processor-based VM-execution controls, and the secondary processor-based VM-execution controls. These are described in Appendix A.3.1, Appendix A.3.2, and Appendix A.3.3, respectively.

A.3.1 Pin-Based VM-Execution Controls

The IA32_VMX_PINBASED_CTLS MSR (index 481H) reports on the allowed settings of **most** of the pin-based VM-execution controls (see Section 24.6.1):

- Bits 31:0 indicate the **allowed 0-settings** of these controls. VM entry allows control X (bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the pin-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 2, and 4; the corresponding bits of the IA32_VMX_PINBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  - If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any pin-based VM-execution control in the default1 class is 0.
  - If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (see below) reports which of the pin-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the **allowed 1-settings** of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (index 48DH) reports on the allowed settings of **all** of the pin-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the pin-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_PINBASED_CTLS MSR. (The IA32_VMX_TRUE_PINBASED_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_TRUE_PINBASED_CTLS MSR. Assuming that software knows that the default1 class of pin-based VM-execution controls contains bits 1, 2, and 4, there is no need for software to consult the IA32_VMX_PINBASED_CTLS MSR.

A.3.2 Primary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the allowed settings of **most** of the primary processor-based VM-execution controls (see Section 24.6.2):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary processor-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the primary processor-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 4–6, 8, 13–16, and 26; the corresponding bits of the IA32_VMX_PROCBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:
  - If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any of the primary processor-based VM-execution controls in the default1 class is 0.
  - If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (see below) reports which of the primary processor-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (index 48EH) reports on the allowed settings of **all** of the primary processor-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary processor-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32_VMX_PROCBASED_CTLS MSR. (The IA32_VMX_TRUE_PROCBASED_CTLS MSR is not supported.)

- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the processor-based VM-execution controls is contained in the IA32_VMX_TRUE_PROCBASED_CTLS MSR. Assuming that software knows that the default1 class of processor-based VM-execution controls contains bits 1, 4–6, 8, 13–16, and 26, there is no need for software to consult the IA32_VMX_PROCBASED_CTLS MSR.

A.3.3 Secondary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH) reports on the allowed settings of the secondary processor-based VM-execution controls (see Section 24.6.2). VM entries perform the following checks:

- Bits 31:0 indicate the allowed 0-settings of these controls. These bits are always 0. This fact indicates that VM entry allows each bit of the secondary processor-based VM-execution controls to be 0 (reserved bits must be 0)
- Bits 63:32 indicate the allowed 1-settings of these controls; the 1-setting is not allowed for any reserved bit. VM entry allows control X (bit X of the secondary processor-based VM-execution controls) to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X and the "activate secondary controls" primary processor-based VM-execution control are both 1.

The IA32_VMX_PROCBASED_CTLS2 MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1).

A.4 VM-EXIT CONTROLS

The IA32_VMX_EXIT_CTLS MSR (index 483H) reports on the allowed settings of **most** of the VM-exit controls (see Section 24.7.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-exit controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-exit controls in the default1 class (see Appendix A.2). These are bits 0–8, 10, 11, 13, 14, 16, and 17; the corresponding bits of the IA32_VMX_EXIT_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-exit control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (see below) reports which of the VM-exit controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (index 48FH) reports on the allowed settings of **all** of the VM-exit controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-exit controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_EXIT_CTLS MSR. (The IA32_VMX_TRUE_EXIT_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_TRUE_EXIT_CTLS MSR. Assuming that software knows that the default1 class of VM-exit controls contains bits 0–8, 10, 11, 13, 14, 16, and 17, there is no need for software to consult the IA32_VMX_EXIT_CTLS MSR.

A.5 VM-ENTRY CONTROLS

The IA32_VMX_ENTRY_CTLS MSR (index 484H) reports on the allowed settings of **most** of the VM-entry controls (see Section 24.8.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-entry controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-entry controls in the default1 class (see Appendix A.2). These are bits 0–8 and 12; the corresponding bits of the IA32_VMX_ENTRY_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-entry control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (see below) reports which of the VM-entry controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X is 1 in the VM-entry controls and bit 32+X is 0 in this MSR.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (index 490H) reports on the allowed settings of **all** of the VM-entry controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-entry controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_ENTRY_CTLS MSR. (The IA32_VMX_TRUE_ENTRY_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_TRUE_ENTRY_CTLS MSR. Assuming that software knows that the default1 class of VM-entry controls contains bits 0–8 and 12, there is no need for software to consult the IA32_VMX_ENTRY_CTLS MSR.

A.6 MISCELLANEOUS DATA

The IA32_VMX_MISC MSR (index 485H) consists of the following fields:

- Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.
- If bit 5 is read as 1, VM exits store the value of IA32_EFER.LMA into the "IA-32e mode guest" VM-entry control; see Section 27.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the "unrestricted guest" VM-execution control.
- Bits 8:6 report, as a bitmap, the activity states supported by the implementation:

  — Bit 6 reports (if set) the support for activity state 1 (HLT).

  — Bit 7 reports (if set) the support for activity state 2 (shutdown).

  — Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).

  If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).
- If bit 14 is read as 1, Intel® Processor Trace (Intel PT) can be used in VMX operation. If the processor supports Intel PT but does not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see "VMXON—Enter VMX Operation" in Chapter 30); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.

- If bit 15 is read as 1, the RDMSR instruction can be used in system-management mode (SMM) to read the IA32_SMBASE MSR (MSR address 9EH). See Section 34.15.6.3.
- Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear).
- Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32_VMX_MISC is N, then 512 * (N + 1) is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
- If bit 28 is read as 1, bit 2 of the IA32_SMM_MONITOR_CTL can be set to 1. VMXOFF unblocks SMIs unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 34.14.4).
- If bit 29 is read as 1, software can use VMWRITE to write to any supported field in the VMCS; otherwise, VMWRITE cannot be used to modify VM-exit information fields.
- If bit 30 is read as 1, VM entry allows injection of a software interrupt, software exception, or privileged software exception with an instruction length of 0.
- Bits 63:32 report the 32-bit MSEG revision identifier used by the processor.
- Bits 13:9 and bit 31 are reserved and are read as 0.

A.7 VMX-FIXED BITS IN CR0

The IA32_VMX_CR0_FIXED0 MSR (index 486H) and IA32_VMX_CR0_FIXED1 MSR (index 487H) indicate how bits in CR0 may be set in VMX operation. They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit is also 1 in IA32_VMX_CR0_FIXED1; if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit is also 0 in IA32_VMX_CR0_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR0_FIXED0 and 1 in IA32_VMX_CR0_FIXED1).

A.8 VMX-FIXED BITS IN CR4

The IA32_VMX_CR4_FIXED0 MSR (index 488H) and IA32_VMX_CR4_FIXED1 MSR (index 489H) indicate how bits in CR4 may be set in VMX operation. They report on bits in CR4 that are allowed to be 0 and 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit of CR4 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit of CR4 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit is also 1 in IA32_VMX_CR4_FIXED1; if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit is also 0 in IA32_VMX_CR4_FIXED0. Thus, each bit in CR4 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR4_FIXED0 and 1 in IA32_VMX_CR4_FIXED1).

A.9 VMCS ENUMERATION

The IA32_VMX_VMCS_ENUM MSR (index 48AH) provides information to assist software in enumerating fields in the VMCS.

As noted in Section 24.11.2, each field in the VMCS is associated with a 32-bit encoding which is structured as follows:

- Bits 31:15 are reserved (must be 0).
- Bits 14:13 indicate the field’s width.
- Bit 12 is reserved (must be 0).
- Bits 11:10 indicate the field’s type.

- Bits 9:1 is an index field that distinguishes different fields with the same width and type.
- Bit 0 indicates access type.

IA32_VMX_VMCS_ENUM indicates to software the highest index value used in the encoding of any field supported by the processor:

- Bits 9:1 contain the highest index value used for any VMCS encoding.
- Bit 0 and bits 63:10 are reserved and are read as 0.

A.10 VPID AND EPT CAPABILITIES

The IA32_VMX_EPT_VPID_CAP MSR (index 48CH) reports information about the capabilities of the logical processor with regard to virtual-processor identifiers (VPIDs, Section 28.1) and extended page tables (EPT, Section 28.2):

- If bit 0 is read as 1, the processor supports execute-only translations by EPT. This support allows software to configure EPT paging-structure entries in which bits 1:0 are clear (indicating that data accesses are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).[1]
- Bit 6 indicates support for a page-walk length of 4.
- If bit 8 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be uncacheable (UC); see Section 24.6.11.
- If bit 14 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be write-back (WB).
- If bit 16 is read as 1, the logical processor allows software to configure a EPT PDE to map a 2-Mbyte page (by setting bit 7 in the EPT PDE).
- If bit 17 is read as 1, the logical processor allows software to configure a EPT PDPTE to map a 1-Gbyte page (by setting bit 7 in the EPT PDPTE).
- Support for the INVEPT instruction (see Chapter 30 and Section 28.3.3.1).
  - — If bit 20 is read as 1, the INVEPT instruction is supported.
  - — If bit 25 is read as 1, the single-context INVEPT type is supported.
  - — If bit 26 is read as 1, the all-context INVEPT type is supported.
- If bit 21 is read as 1, accessed and dirty flags for EPT are supported (see Section 28.2.4).
- If bit 22 is read as 1, the processor reports advanced VM-exit information for EPT violations (see Section 27.2.1). This reporting is done only if this bit is read as 1.
- Support for the INVVPID instruction (see Chapter 30 and Section 28.3.3.1).
  - — If bit 32 is read as 1, the INVVPID instruction is supported.
  - — If bit 40 is read as 1, the individual-address INVVPID type is supported.
  - — If bit 41 is read as 1, the single-context INVVPID type is supported.
  - — If bit 42 is read as 1, the all-context INVVPID type is supported.
  - — If bit 43 is read as 1, the single-context-retaining-globals INVVPID type is supported.
- Bits 5:1, bit 7, bits 13:9, bit 15, bits 19:18, bits 24:23, bits 31:27, bits 39:33, and bits 63:44 are reserved and are read as 0.

1. If the "mode-based execute control for EPT" VM-execution control is 1, setting bit 0 indicates also that software may also configure EPT paging-structure entries in which bits 1:0 are both clear and in which bit 10 is set (indicating a translation that can be used to fetch instructions from a supervisor-mode linear address or a user-mode linear address).

The IA32_VMX_EPT_VPID_CAP MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and that support either the 1-setting of the "enable EPT" VM-execution control (only if bit 33 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1) or the 1-setting of the "enable VPID" VM-execution control (only if bit 37 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).

A.11 VM FUNCTIONS

The IA32_VMX_VMFUNC MSR (index 491H) reports on the allowed settings of the VM-function controls (see Section 24.6.14). VM entry allows bit X of the VM-function controls to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if bit X of the VM-function controls, the "activate secondary controls" primary processor-based VM-execution control, and the "enable VM functions" secondary processor-based VM-execution control are all 1.

The IA32_VMX_VMFUNC MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and the 1-setting of the "enable VM functions" secondary processor-based VM-execution control (only if bit 45 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).

29.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: Update comments for IA32_SGXLEPUBKEYHASH0, IA32_SGXLEPUBKEYHASH1, IA32_SGXLEPUBKEYHASH2, and IA32_SGXLEPUBKEYHASH3 MSRs. Updated MSR_SGXOWNEREPOCH0 and MSR_SGXOWNEREPOCH1 to indicate they are write only. Updated MSR_BR_DETECT_CTRL with additional information regarding bits 17:8. Updated FREEZE_WHILE_SMM name.

CHAPTER 2
MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_85H	Future Intel® Xeon Phi™ Processor based on Knights Mill microarchitecture
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
06_66H	Future Intel® Core™ processors based on Cannon Lake microarchitecture
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture
06_55H	Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Intel microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series

Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number 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_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
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily_DisplayModel = 05_09H and SteppingID = 0

2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_". Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific.

Code that accesses a machine specified MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF_DM" (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYADDR" in Table 2-2. "MAXPHYADDR" is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

Table 2-2. IA-32 Architectural MSRs

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 2.22, "MSRs in Pentium Processors."	**Pentium Processor (05_01H)**
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 2.22, "MSRs in Pentium Processors."	DF_DM = 05_01H
6H	6	IA32_MONITOR_FILTER_SIZE	See Section 8.10.5, "Monitor/Mwait Address Range Determination."	0F_03H
10H	16	IA32_TIME_STAMP_ COUNTER (TSC)	See Section 17.17, "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.	
		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)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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
		1	Enable VMX inside SMX operation (R/WL): This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
		2	Enable VMX outside SMX operation (R/WL): This bit enables VMX for system executive that do not require SMX. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
		7:3	Reserved	
		14:8	SENTER Local Function Enables (R/WL): When set, each bit in the field represents an enable control for a corresponding SENTER function. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		15	SENTER Global Enable (R/WL): This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		16	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		17	SGX Launch Control Enable (R/WL): This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
		18	SGX Global Enable (R/WL): This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		19	Reserved	
		20	LMCE On (R/WL): When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
		63:21	Reserved	
3BH	59	IA32_TSC_ADJUST	Per Logical Processor TSC Adjust (R/Write to clear)	If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
		63:0	**THREAD_ADJUST:** Local offset value of the IA32_TSC for a logical processor. Reset value is Zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	
79H	121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 9.11.6, "Microcode Update Loader." A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
8BH	139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	BIOS Update Signature (RO) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
		31:0	Reserved	
		63:32	It is recommended that this field be pre-loaded with 0 prior to executing CPUID. If the field remains 0 following the execution of CPUID; this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	**C0_MCNT: C0 TSC Frequency Clock Count** Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_APERF.	
E8H	232	IA32_APERF	Actual Performance Clock Counter (R/Write to clear).	If CPUID.06H: ECX[0] = 1
		63:0	**C0_ACNT: C0 Actual Frequency Clock Count** Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	06_01H
		7:0	VCNT: The number of variable memory type ranges in the processor.	
		8	Fixed range MTRRs are supported when set.	
		9	Reserved.	
		10	WC Supported when set.	
		11	SMRR Supported when set.	
		63:12	Reserved.	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector.	
		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	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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.	
		26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.	06_3EH
		27	MCG_LMCE_P: Indicates that the processor support extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
		63:28	Reserved.	
17AH	378	IA32_MCG_STATUS (MCG_STATUS)	Global Machine Check Status (R/W0)	06_01H
		0	RIPV. Restart IP valid	06_01H
		1	EIPV. Error IP valid	06_01H
		2	MCIP. Machine check in progress	06_01H
		3	LMCE_S.	If IA32_MCG_CAP.LMCE_P[27] =1
		63:4	Reserved.	
17BH	379	IA32_MCG_CTL (MCG_CTL)	Global Machine Check Control (R/W)	If IA32_MCG_CAP.CTL_P[8] =1
180H-185H	384-389	Reserved		06_0EH[1]
186H	390	IA32_PERFEVTSEL0 (PERFEVTSEL0)	Performance Event Select Register 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
		7:0	Event Select: Selects a performance event logic unit.	
		15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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.	
		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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		32	IDA Engage. (R/W) When set to 1: disengages IDA	06_0FH (Mobile only)
		63:33	Reserved.	
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.7.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1
		0	Extended On-Demand Clock Modulation Duty Cycle:	If CPUID.06H:EAX[5] = 1
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
		63:5	Reserved.	
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.7.2, "Thermal Monitor."	If CPUID.01H:EDX[22] = 1
		0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		7:5	Reserved.	
		14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
		15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
		23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved.	
19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.7.2, "Thermal Monitor"	If CPUID.01H:EDX[22] = 1
		0	Thermal Status (RO):	If CPUID.01H:EDX[22] = 1
		1	Thermal Status Log (R/W):	If CPUID.01H:EDX[22] = 1
		2	PROCHOT # or FORCEPR# event (RO)	If CPUID.01H:EDX[22] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		3	PROCHOT # or FORCEPR# log (R/WC0)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WC0)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #2 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WC0)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved.	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1
		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved.	
1A0H	416	IA32_MISC_ENABLE	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.	
		0	**Fast-Strings Enable** When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default); when clear, fast-strings are disabled.	0F_0H
		2:1	Reserved.	
		3	**Automatic Thermal Control Circuit Enable (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated. The default value of this field varies with product . See respective tables where default value is listed.	0F_0H

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		6:4	Reserved	
		7	**Performance Monitoring Available (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled	0F_0H
		10:8	Reserved.	
		11	**Branch Trace Storage Unavailable (RO)** 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported	0F_0H
		12	**Processor Event Based Sampling (PEBS) Unavailable (RO)** 1 = PEBS is not supported; 0 = PEBS is supported.	06_0FH
		15:13	Reserved.	
		16	**Enhanced Intel SpeedStep Technology Enable (R/W)** 0= Enhanced Intel SpeedStep Technology disabled 1 = Enhanced Intel SpeedStep Technology enabled	If CPUID.01H: ECX[7] =1
		17	Reserved.	
		18	**ENABLE MONITOR FSM (R/W)** When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
		21:19	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		22	**Limit CPUID Maxval (R/W)** When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2. BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2. Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported. Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception. Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.	0F_03H
		23	**xTPR Message Disable (R/W)** When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.	if CPUID.01H:ECX[14] = 1
		33:24	Reserved.	
		34	**XD Bit Disable (R/W)** When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location, if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.	if CPUID.80000001H:EDX[20] = 1
		63:35	Reserved.	
1B0H	432	IA32_ENERGY_PERF_BIAS	Performance Energy Bias Hint (R/W)	if CPUID.6H:ECX[3] = 1
		3:0	Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.	
		63:4	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg Thermal Status (RO):	
		1	Pkg Thermal Status Log (R/W):	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WC0)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status log (R/WC0)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 log (R/WC0)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 log (R/WC0)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation log (R/WC0)	
		15:12	Reserved.	
		22:16	Pkg Digital Readout (RO)	
		63:23	Reserved.	
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved.	
		4	Pkg Overheat Interrupt Enable	
		7:5	Reserved.	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
		22:16	Pkg Threshold #2 Value	
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:25	Reserved.	
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved.	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
		15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
		63:16	Reserved.	
1F2H	498	IA32_SMRR_PHYSBASE	**SMRR Base Address (Writeable only in SMM)** Base address of SMM memory range.	If IA32_MTRRCAP.SMRR[11] = 1
		7:0	Type. Specifies memory type of the range.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11:8	Reserved.	
		31:12	**PhysBase.** SMRR physical Base Address.	
		63:32	Reserved.	
1F3H	499	IA32_SMRR_PHYSMASK	**SMRR Range Mask. (Writeable only in SMM)** Range Mask of SMM memory range.	If IA32_MTRRCAP[SMRR] = 1
		10:0	Reserved.	
		11	**Valid** Enable range mask.	
		31:12	**PhysMask** SMRR address range mask.	
		63:32	Reserved.	
1F8H	504	IA32_PLATFORM_DCA_CAP	DCA Capability (R)	If CPUID.01H: ECX[18] = 1
1F9H	505	IA32_CPU_DCA_CAP	If set, CPU supports Prefetch-Hint type.	If CPUID.01H: ECX[18] = 1
1FAH	506	IA32_DCA_0_CAP	DCA type 0 Status and Control register.	If CPUID.01H: ECX[18] = 1
		0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
		2:1	TRANSACTION	
		6:3	DCA_TYPE	
		10:7	DCA_QUEUE_SIZE	
		12:11	Reserved.	
		16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
		23:17	Reserved.	
		24	SW_BLOCK: SW can request DCA block by setting this bit.	
		25	Reserved.	
		26	HW_BLOCK: Set when DCA is blocked by HW (e.g. CR0.CD = 1).	
		31:27	Reserved.	
200H	512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	See Section 11.11.2.3, "Variable Range MTRRs."	If CPUID.01H: EDX.MTRR[12] =1
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If CPUID.01H: EDX.MTRR[12] =1
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If CPUID.01H: EDX.MTRR[12] =1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If CPUID.01H: EDX.MTRR[12] =1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If CPUID.01H: EDX.MTRR[12] =1
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If CPUID.01H: EDX.MTRR[12] =1
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If CPUID.01H: EDX.MTRR[12] =1
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If CPUID.01H: EDX.MTRR[12] =1
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If CPUID.01H: EDX.MTRR[12] =1
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If CPUID.01H: EDX.MTRR[12] =1
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If CPUID.01H: EDX.MTRR[12] =1
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If CPUID.01H: EDX.MTRR[12] =1
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If CPUID.01H: EDX.MTRR[12] =1
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If CPUID.01H: EDX.MTRR[12] =1
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If CPUID.01H: EDX.MTRR[12] =1
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If CPUID.01H: EDX.MTRR[12] =1
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	if IA32_MTRRCAP[7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	if IA32_MTRRCAP[7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	if IA32_MTRRCAP[7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	if IA32_MTRRCAP[7:0] > 9
250H	592	IA32_MTRR_FIX64K_00000	MTRRfix64K_00000	If CPUID.01H: EDX.MTRR[12] =1
258H	600	IA32_MTRR_FIX16K_80000	MTRRfix16K_80000	If CPUID.01H: EDX.MTRR[12] =1
259H	601	IA32_MTRR_FIX16K_A0000	MTRRfix16K_A0000	If CPUID.01H: EDX.MTRR[12] =1
268H	616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)	See Section 11.11.2.2, "Fixed Range MTRRs."	If CPUID.01H: EDX.MTRR[12] =1
269H	617	IA32_MTRR_FIX4K_C8000	MTRRfix4K_C8000	If CPUID.01H: EDX.MTRR[12] =1
26AH	618	IA32_MTRR_FIX4K_D0000	MTRRfix4K_D0000	If CPUID.01H: EDX.MTRR[12] =1
26BH	619	IA32_MTRR_FIX4K_D8000	MTRRfix4K_D8000	If CPUID.01H: EDX.MTRR[12] =1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
26CH	620	IA32_MTRR_FIX4K_E0000	MTRRfix4K_E0000	If CPUID.01H: EDX.MTRR[12] =1
26DH	621	IA32_MTRR_FIX4K_E8000	MTRRfix4K_E8000	If CPUID.01H: EDX.MTRR[12] =1
26EH	622	IA32_MTRR_FIX4K_F0000	MTRRfix4K_F0000	If CPUID.01H: EDX.MTRR[12] =1
26FH	623	IA32_MTRR_FIX4K_F8000	MTRRfix4K_F8000	If CPUID.01H: EDX.MTRR[12] =1
277H	631	IA32_PAT	IA32_PAT (R/W)	If CPUID.01H: EDX.MTRR[16] =1
		2:0	PA0	
		7:3	Reserved.	
		10:8	PA1	
		15:11	Reserved.	
		18:16	PA2	
		23:19	Reserved.	
		26:24	PA3	
		31:27	Reserved.	
		34:32	PA4	
		39:35	Reserved.	
		42:40	PA5	
		47:43	Reserved.	
		50:48	PA6	
		55:51	Reserved.	
		58:56	PA7	
		63:59	Reserved.	
280H	640	IA32_MC0_CTL2	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_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
283H	643	IA32_MC3_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6
287H	647	IA32_MC7_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
28CH	652	IA32_MC12_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
292H	658	IA32_MC18_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21
296H	662	IA32_MC22_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
29BH	667	IA32_MC27_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
2FFH	767	IA32_MTRR_DEF_TYPE	MTRRdefType (R/W)	If CPUID.01H: EDX.MTRR[12] =1
		2:0	Default Memory Type	
		9:3	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved.	
309H	777	IA32_FIXED_CTR0 (MSR_PERF_FIXED_CTR0)	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1 (MSR_PERF_FIXED_CTR1)	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2 (MSR_PERF_FIXED_CTR2)	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref	If CPUID.0AH: EDX[4:0] > 2
345H	837	IA32_PERF_CAPABILITIES	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.	
		63:14	Reserved.	
38DH	909	IA32_FIXED_CTR_CTRL	Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0.	
		1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
		2	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		3	EN0_PMI: Enable PMI when fixed counter 0 overflows.	
		4	EN1_OS: Enable Fixed Counter 1to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1to count while CPL > 0.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		6	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
		10	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
		63:12	Reserved.	
38EH	910	IA32_PERF_GLOBAL_STATUS	Global Performance Counter Status (RO)	If CPUID.0AH: EAX[7:0] > 0
		0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
		1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
		2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
		3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3
		31:4	Reserved.	
		32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
		33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1
		34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
		54:35	Reserved.	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer was completely filled.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		58	LBR_Frz: LBRs are frozen due to ▪ IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1, ▪ The LBR stack overflowed	If CPUID.0AH: EAX[7:0] > 3
		59	CTR_Frz: Performance counters in the core PMU are frozen due to ▪ IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1, ▪ one or more core PMU counters overflowed.	If CPUID.0AH: EAX[7:0] > 3
		60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation intel SGX to protect an enclave.	If CPUID.(EAX=07H, ECX=0):EBX[2] = 1
		61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.0AH: EAX[7:0] > 2
		62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.0AH: EAX[7:0] > 0
		63	CondChgd: status bits of this register has changed.	If CPUID.0AH: EAX[7:0] > 0
38FH	911	IA32_PERF_GLOBAL_CTRL	Global Performance Counter Control (R/W) Counter increments while the result of ANDing respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.	If CPUID.0AH: EAX[7:0] > 0
		0	EN_PMC0	If CPUID.0AH: EAX[15:8] > 0
		1	EN_PMC1	If CPUID.0AH: EAX[15:8] > 1
		2	EN_PMC2	If CPUID.0AH: EAX[15:8] > 2
		n	EN_PMCn	If CPUID.0AH: EAX[15:8] > n
		31:n+1	Reserved.	
		32	EN_FIXED_CTR0	If CPUID.0AH: EDX[4:0] > 0
		33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2
		63:35	Reserved.	
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Global Performance Counter Overflow Control (R/W)	If CPUID.0AH: EAX[7:0] > 0 && CPUID.0AH: EAX[7:0] <= 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		60:56	Reserved.	
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Global Performance Counter Overflow Reset Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA[8] = 1
		57:56	Reserved.	
		58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to Clear ASCI bit.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Global Performance Counter Overflow Set Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		1	Set 1 to cause Ovf_PMC1 = 1	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to cause Ovf_PMC2 = 1	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to cause Ovf_PMCn = 1	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.0AH: EAX[7:0] > 3
		34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.0AH: EAX[7:0] > 3
		54:35	Reserved.	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.0AH: EAX[7:0] > 3
		57:56	Reserved.	
		58	Set 1 to cause LBR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to cause CTR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to cause ASCI = 1.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.0AH: EAX[7:0] > 3
		62	Set 1 to cause OvfBuf = 1.	If CPUID.0AH: EAX[7:0] > 3
		63	Reserved	
392H	914	IA32_PERF_GLOBAL_INUSE	Indicator of core perfmon interface is in use (RO)	If CPUID.0AH: EAX[7:0] > 3
		0	IA32_PERFEVTSEL0 in use	
		1	IA32_PERFEVTSEL1 in use	If CPUID.0AH: EAX[15:8] > 1
		2	IA32_PERFEVTSEL2 in use	If CPUID.0AH: EAX[15:8] > 2
		n	IA32_PERFEVTSELn in use	If CPUID.0AH: EAX[15:8] > n
		31:n+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)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	Enable PEBS on IA32_PMC0.	06_0FH
		3:1	Reserved or Model specific.	
		31:4	Reserved.	
		35:32	Reserved or Model specific.	
		63:36	Reserved.	
400H	1024	IA32_MC0_CTL	MC0_CTL	If IA32_MCG_CAP.CNT >0
401H	1025	IA32_MC0_STATUS	MC0_STATUS	If IA32_MCG_CAP.CNT >0
402H	1026	IA32_MC0_ADDR[1]	MC0_ADDR	If IA32_MCG_CAP.CNT >0
403H	1027	IA32_MC0_MISC	MC0_MISC	If IA32_MCG_CAP.CNT >0
404H	1028	IA32_MC1_CTL	MC1_CTL	If IA32_MCG_CAP.CNT >1
405H	1029	IA32_MC1_STATUS	MC1_STATUS	If IA32_MCG_CAP.CNT >1
406H	1030	IA32_MC1_ADDR[2]	MC1_ADDR	If IA32_MCG_CAP.CNT >1
407H	1031	IA32_MC1_MISC	MC1_MISC	If IA32_MCG_CAP.CNT >1
408H	1032	IA32_MC2_CTL	MC2_CTL	If IA32_MCG_CAP.CNT >2
409H	1033	IA32_MC2_STATUS	MC2_STATUS	If IA32_MCG_CAP.CNT >2
40AH	1034	IA32_MC2_ADDR[1]	MC2_ADDR	If IA32_MCG_CAP.CNT >2
40BH	1035	IA32_MC2_MISC	MC2_MISC	If IA32_MCG_CAP.CNT >2
40CH	1036	IA32_MC3_CTL	MC3_CTL	If IA32_MCG_CAP.CNT >3
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	If IA32_MCG_CAP.CNT >3
40EH	1038	IA32_MC3_ADDR[1]	MC3_ADDR	If IA32_MCG_CAP.CNT >3
40FH	1039	IA32_MC3_MISC	MC3_MISC	If IA32_MCG_CAP.CNT >3
410H	1040	IA32_MC4_CTL	MC4_CTL	If IA32_MCG_CAP.CNT >4
411H	1041	IA32_MC4_STATUS	MC4_STATUS	If IA32_MCG_CAP.CNT >4
412H	1042	IA32_MC4_ADDR[1]	MC4_ADDR	If IA32_MCG_CAP.CNT >4
413H	1043	IA32_MC4_MISC	MC4_MISC	If IA32_MCG_CAP.CNT >4
414H	1044	IA32_MC5_CTL	MC5_CTL	If IA32_MCG_CAP.CNT >5
415H	1045	IA32_MC5_STATUS	MC5_STATUS	If IA32_MCG_CAP.CNT >5
416H	1046	IA32_MC5_ADDR[1]	MC5_ADDR	If IA32_MCG_CAP.CNT >5
417H	1047	IA32_MC5_MISC	MC5_MISC	If IA32_MCG_CAP.CNT >5
418H	1048	IA32_MC6_CTL	MC6_CTL	If IA32_MCG_CAP.CNT >6
419H	1049	IA32_MC6_STATUS	MC6_STATUS	If IA32_MCG_CAP.CNT >6
41AH	1050	IA32_MC6_ADDR[1]	MC6_ADDR	If IA32_MCG_CAP.CNT >6
41BH	1051	IA32_MC6_MISC	MC6_MISC	If IA32_MCG_CAP.CNT >6
41CH	1052	IA32_MC7_CTL	MC7_CTL	If IA32_MCG_CAP.CNT >7
41DH	1053	IA32_MC7_STATUS	MC7_STATUS	If IA32_MCG_CAP.CNT >7
41EH	1054	IA32_MC7_ADDR[1]	MC7_ADDR	If IA32_MCG_CAP.CNT >7
41FH	1055	IA32_MC7_MISC	MC7_MISC	If IA32_MCG_CAP.CNT >7

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
420H	1056	IA32_MC8_CTL	MC8_CTL	If IA32_MCG_CAP.CNT >8
421H	1057	IA32_MC8_STATUS	MC8_STATUS	If IA32_MCG_CAP.CNT >8
422H	1058	IA32_MC8_ADDR[1]	MC8_ADDR	If IA32_MCG_CAP.CNT >8
423H	1059	IA32_MC8_MISC	MC8_MISC	If IA32_MCG_CAP.CNT >8
424H	1060	IA32_MC9_CTL	MC9_CTL	If IA32_MCG_CAP.CNT >9
425H	1061	IA32_MC9_STATUS	MC9_STATUS	If IA32_MCG_CAP.CNT >9
426H	1062	IA32_MC9_ADDR[1]	MC9_ADDR	If IA32_MCG_CAP.CNT >9
427H	1063	IA32_MC9_MISC	MC9_MISC	If IA32_MCG_CAP.CNT >9
428H	1064	IA32_MC10_CTL	MC10_CTL	If IA32_MCG_CAP.CNT >10
429H	1065	IA32_MC10_STATUS	MC10_STATUS	If IA32_MCG_CAP.CNT >10
42AH	1066	IA32_MC10_ADDR[1]	MC10_ADDR	If IA32_MCG_CAP.CNT >10
42BH	1067	IA32_MC10_MISC	MC10_MISC	If IA32_MCG_CAP.CNT >10
42CH	1068	IA32_MC11_CTL	MC11_CTL	If IA32_MCG_CAP.CNT >11
42DH	1069	IA32_MC11_STATUS	MC11_STATUS	If IA32_MCG_CAP.CNT >11
42EH	1070	IA32_MC11_ADDR[1]	MC11_ADDR	If IA32_MCG_CAP.CNT >11
42FH	1071	IA32_MC11_MISC	MC11_MISC	If IA32_MCG_CAP.CNT >11
430H	1072	IA32_MC12_CTL	MC12_CTL	If IA32_MCG_CAP.CNT >12
431H	1073	IA32_MC12_STATUS	MC12_STATUS	If IA32_MCG_CAP.CNT >12
432H	1074	IA32_MC12_ADDR[1]	MC12_ADDR	If IA32_MCG_CAP.CNT >12
433H	1075	IA32_MC12_MISC	MC12_MISC	If IA32_MCG_CAP.CNT >12
434H	1076	IA32_MC13_CTL	MC13_CTL	If IA32_MCG_CAP.CNT >13
435H	1077	IA32_MC13_STATUS	MC13_STATUS	If IA32_MCG_CAP.CNT >13
436H	1078	IA32_MC13_ADDR[1]	MC13_ADDR	If IA32_MCG_CAP.CNT >13
437H	1079	IA32_MC13_MISC	MC13_MISC	If IA32_MCG_CAP.CNT >13
438H	1080	IA32_MC14_CTL	MC14_CTL	If IA32_MCG_CAP.CNT >14
439H	1081	IA32_MC14_STATUS	MC14_STATUS	If IA32_MCG_CAP.CNT >14
43AH	1082	IA32_MC14_ADDR[1]	MC14_ADDR	If IA32_MCG_CAP.CNT >14
43BH	1083	IA32_MC14_MISC	MC14_MISC	If IA32_MCG_CAP.CNT >14
43CH	1084	IA32_MC15_CTL	MC15_CTL	If IA32_MCG_CAP.CNT >15
43DH	1085	IA32_MC15_STATUS	MC15_STATUS	If IA32_MCG_CAP.CNT >15
43EH	1086	IA32_MC15_ADDR[1]	MC15_ADDR	If IA32_MCG_CAP.CNT >15
43FH	1087	IA32_MC15_MISC	MC15_MISC	If IA32_MCG_CAP.CNT >15
440H	1088	IA32_MC16_CTL	MC16_CTL	If IA32_MCG_CAP.CNT >16
441H	1089	IA32_MC16_STATUS	MC16_STATUS	If IA32_MCG_CAP.CNT >16
442H	1090	IA32_MC16_ADDR[1]	MC16_ADDR	If IA32_MCG_CAP.CNT >16
443H	1091	IA32_MC16_MISC	MC16_MISC	If IA32_MCG_CAP.CNT >16
444H	1092	IA32_MC17_CTL	MC17_CTL	If IA32_MCG_CAP.CNT >17

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
445H	1093	IA32_MC17_STATUS	MC17_STATUS	If IA32_MCG_CAP.CNT >17
446H	1094	IA32_MC17_ADDR[1]	MC17_ADDR	If IA32_MCG_CAP.CNT >17
447H	1095	IA32_MC17_MISC	MC17_MISC	If IA32_MCG_CAP.CNT >17
448H	1096	IA32_MC18_CTL	MC18_CTL	If IA32_MCG_CAP.CNT >18
449H	1097	IA32_MC18_STATUS	MC18_STATUS	If IA32_MCG_CAP.CNT >18
44AH	1098	IA32_MC18_ADDR[1]	MC18_ADDR	If IA32_MCG_CAP.CNT >18
44BH	1099	IA32_MC18_MISC	MC18_MISC	If IA32_MCG_CAP.CNT >18
44CH	1100	IA32_MC19_CTL	MC19_CTL	If IA32_MCG_CAP.CNT >19
44DH	1101	IA32_MC19_STATUS	MC19_STATUS	If IA32_MCG_CAP.CNT >19
44EH	1102	IA32_MC19_ADDR[1]	MC19_ADDR	If IA32_MCG_CAP.CNT >19
44FH	1103	IA32_MC19_MISC	MC19_MISC	If IA32_MCG_CAP.CNT >19
450H	1104	IA32_MC20_CTL	MC20_CTL	If IA32_MCG_CAP.CNT >20
451H	1105	IA32_MC20_STATUS	MC20_STATUS	If IA32_MCG_CAP.CNT >20
452H	1106	IA32_MC20_ADDR[1]	MC20_ADDR	If IA32_MCG_CAP.CNT >20
453H	1107	IA32_MC20_MISC	MC20_MISC	If IA32_MCG_CAP.CNT >20
454H	1108	IA32_MC21_CTL	MC21_CTL	If IA32_MCG_CAP.CNT >21
455H	1109	IA32_MC21_STATUS	MC21_STATUS	If IA32_MCG_CAP.CNT >21
456H	1110	IA32_MC21_ADDR[1]	MC21_ADDR	If IA32_MCG_CAP.CNT >21
457H	1111	IA32_MC21_MISC	MC21_MISC	If IA32_MCG_CAP.CNT >21
458H		IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H		IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH		IA32_MC22_ADDR[1]	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH		IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH		IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH		IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH		IA32_MC23_ADDR[1]	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH		IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H		IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H		IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H		IA32_MC24_ADDR[1]	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H		IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H		IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H		IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H		IA32_MC25_ADDR[1]	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H		IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H		IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H		IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
46AH		IA32_MC26_ADDR[1]	MC26_ADDR	If IA32_MCG_CAP.CNT >26
46BH		IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT >26
46CH		IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT >27
46DH		IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT >27
46EH		IA32_MC27_ADDR[1]	MC27_ADDR	If IA32_MCG_CAP.CNT >27
46FH		IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT >27
470H		IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT >28
471H		IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT >28
472H		IA32_MC28_ADDR[1]	MC28_ADDR	If IA32_MCG_CAP.CNT >28
473H		IA32_MC28_MISC	MC28_MISC	If IA32_MCG_CAP.CNT >28
480H	1152	IA32_VMX_BASIC	**Reporting Register of Basic VMX Capabilities (R/O)** See Appendix A.1, "Basic VMX Information."	If CPUID.01H:ECX.[5] = 1
481H	1153	IA32_VMX_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
482H	1154	IA32_VMX_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
483H	1155	IA32_VMX_EXIT_CTLS	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls."	If CPUID.01H:ECX.[5] = 1
484H	1156	IA32_VMX_ENTRY_CTLS	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls."	If CPUID.01H:ECX.[5] = 1
485H	1157	IA32_VMX_MISC	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data."	If CPUID.01H:ECX.[5] = 1
486H	1158	IA32_VMX_CR0_FIXED0	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
487H	1159	IA32_VMX_CR0_FIXED1	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
488H	1160	IA32_VMX_CR4_FIXED0	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
489H	1161	IA32_VMX_CR4_FIXED1	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
48AH	1162	IA32_VMX_VMCS_ENUM	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration."	If CPUID.01H:ECX.[5] = 1
48BH	1163	IA32_VMX_PROCBASED_CTLS2	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3.3, "Secondary Processor-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63])
48CH	1164	IA32_VMX_EPT_VPID_CAP	**Capability Reporting Register of EPT and VPID (R/O)** See Appendix A.10, "VPID and EPT Capabilities."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63] && (IA32_VMX_PROCBASED_CTLS2[33] || IA32_VMX_PROCBASED_CTLS2[37]))
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Appendix A.4, "VM-Exit Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Appendix A.5, "VM-Entry Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
491H	1169	IA32_VMX_VMFUNC	**Capability Reporting Register of VM-function Controls (R/O)**	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1
4D0H	1232	IA32_MCG_EXT_CTL	Allows software to signal some MCEs to only 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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
560H	1376	IA32_RTIT_OUTPUT_BASE	**Trace Output Base Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		MAXPHYADDR[3]-1:7	Base physical address	
		63:MAXPHYADDR	**Reserved.**	
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	**Trace Output Mask Pointers Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	**Output Offset.**	
570H	1392	IA32_RTIT_CTL	**Trace Control Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**TraceEn**	
		1	**CYCEn**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		2	**OS**	
		3	**User**	
		4	**PwrEvtEn**	
		5	**FUPonPTW**	
		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**	
		13	**BranchEn**	
		17:14	**MTCFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		18	Reserved, MBZ	
		22:19	**CYCThresh**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		23	Reserved, MBZ	
		27:24	**PSBFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		31:28	Reserved, MBZ	
		35:32	**ADDR0_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		39:36	**ADDR1_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		43:40	**ADDR2_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:44	**ADDR3_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		63:48	Reserved, MBZ.	
571H	1393	IA32_RTIT_STATUS	**Tracing Status Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**FilterEn** (writes ignored)	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)
		1	**ContexEn** (writes ignored)	
		2	**TriggerEn** (writes ignored)	
		3	Reserved	
		4	**Error**	
		5	**Stopped**	
		31:6	Reserved, MBZ	
		48:32	**PacketByteCnt**	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
		63:49	**Reserved**	
572H	1394	IA32_RTIT_CR3_MATCH	**Trace Filter CR3 Match Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match	
580H	1408	IA32_RTIT_ADDR0_A	**Region 0 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
581H	1409	IA32_RTIT_ADDR0_B	**Region 0 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
582H	1410	IA32_RTIT_ADDR1_A	**Region 1 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:48	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	
		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.	
6E0H	1760	IA32_TSC_DEADLINE	**TSC Target of Local APIC's TSC Deadline Mode (R/W)**	If CPUID.01H:ECX.[24] = 1
770H	1904	IA32_PM_ENABLE	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[7] = 1
		0	**HWP_ENABLE (R/W1-Once)** See Section 14.4.2, "Enabling HWP"	If CPUID.06H:EAX.[7] = 1
		63:1	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
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
		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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		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.	
777H	1911	IA32_HWP_STATUS	**Log bits indicating changes to Guaranteed & excursions to Minimum (R/W)**	If CPUID.06H:EAX.[7] = 1
		0	**Guaranteed_Performance_Change (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		1	Reserved.	
		2	**Excursion_To_Minimum (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved.	
802H	2050	IA32_X2APIC_APICID	**x2APIC ID Register (R/O)** See x2APIC Specification	If CPUID.01H:ECX[21] = 1 && IA32_APIC_BASE.[10] = 1
803H	2051	IA32_X2APIC_VERSION	**x2APIC Version Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
808H	2056	IA32_X2APIC_TPR	**x2APIC Task Priority Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80AH	2058	IA32_X2APIC_PPR	**x2APIC Processor Priority Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80BH	2059	IA32_X2APIC_EOI	**x2APIC EOI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80DH	2061	IA32_X2APIC_LDR	**x2APIC Logical Destination Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80FH	2063	IA32_X2APIC_SIVR	**x2APIC Spurious Interrupt Vector Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
810H	2064	IA32_X2APIC_ISR0	**x2APIC In-Service Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
811H	2065	IA32_X2APIC_ISR1	**x2APIC In-Service Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
812H	2066	IA32_X2APIC_ISR2	**x2APIC In-Service Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
813H	2067	IA32_X2APIC_ISR3	**x2APIC In-Service Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
814H	2068	IA32_X2APIC_ISR4	**x2APIC In-Service Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
815H	2069	IA32_X2APIC_ISR5	**x2APIC In-Service Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
816H	2070	IA32_X2APIC_ISR6	**x2APIC In-Service Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
817H	2071	IA32_X2APIC_ISR7	**x2APIC In-Service Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
818H	2072	IA32_X2APIC_TMR0	**x2APIC Trigger Mode Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
819H	2073	IA32_X2APIC_TMR1	**x2APIC Trigger Mode Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81AH	2074	IA32_X2APIC_TMR2	**x2APIC Trigger Mode Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81BH	2075	IA32_X2APIC_TMR3	**x2APIC Trigger Mode Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81CH	2076	IA32_X2APIC_TMR4	**x2APIC Trigger Mode Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81DH	2077	IA32_X2APIC_TMR5	**x2APIC Trigger Mode Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81EH	2078	IA32_X2APIC_TMR6	**x2APIC Trigger Mode Register Bits 223:192 (R/O)**	If (CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
81FH	2079	IA32_X2APIC_TMR7	**x2APIC Trigger Mode Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
820H	2080	IA32_X2APIC_IRR0	**x2APIC Interrupt Request Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
821H	2081	IA32_X2APIC_IRR1	**x2APIC Interrupt Request Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
822H	2082	IA32_X2APIC_IRR2	**x2APIC Interrupt Request Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
823H	2083	IA32_X2APIC_IRR3	**x2APIC Interrupt Request Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
824H	2084	IA32_X2APIC_IRR4	**x2APIC Interrupt Request Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
825H	2085	IA32_X2APIC_IRR5	**x2APIC Interrupt Request Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
826H	2086	IA32_X2APIC_IRR6	**x2APIC Interrupt Request Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
827H	2087	IA32_X2APIC_IRR7	**x2APIC Interrupt Request Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
828H	2088	IA32_X2APIC_ESR	**x2APIC Error Status Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
82FH	2095	IA32_X2APIC_LVT_CMCI	**x2APIC LVT Corrected Machine Check Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
830H	2096	IA32_X2APIC_ICR	**x2APIC Interrupt Command Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
832H	2098	IA32_X2APIC_LVT_TIMER	**x2APIC LVT Timer Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
833H	2099	IA32_X2APIC_LVT_THERMAL	**x2APIC LVT Thermal Sensor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
834H	2100	IA32_X2APIC_LVT_PMI	**x2APIC LVT Performance Monitor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
835H	2101	IA32_X2APIC_LVT_LINT0	**x2APIC LVT LINT0 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
836H	2102	IA32_X2APIC_LVT_LINT1	**x2APIC LVT LINT1 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
837H	2103	IA32_X2APIC_LVT_ERROR	**x2APIC LVT Error Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
838H	2104	IA32_X2APIC_INIT_COUNT	**x2APIC Initial Count Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
839H	2105	IA32_X2APIC_CUR_COUNT	**x2APIC Current Count Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83EH	2110	IA32_X2APIC_DIV_CONF	**x2APIC Divide Configuration Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83FH	2111	IA32_X2APIC_SELF_IPI	**x2APIC Self IPI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
C80H	3200	IA32_DEBUG_INTERFACE	**Silicon Debug Feature Control (R/W)**	If CPUID.01H:ECX.[11] = 1
		0	**Enable (R/W)** BIOS set 1 to enable Silicon debug features. Default is 0	If CPUID.01H:ECX.[11] = 1
		29:1	Reserved.	
		30	**Lock (R/W)**: If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
		31	**Debug Occurred (R/O)**: This "sticky bit" is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
		63:32	Reserved.	
C81H	3201	IA32_L3_QOS_CFG	**L3 QOS Configuration (R/W)**	If (CPUID.(EAX=10H, ECX=1):ECX.[2] = 1)
		0	**Enable (R/W)** Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode.	
		63:1	Reserved. 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)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	**Enable (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 and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
C8FH	3215	IA32_PQR_ASSOC	**Resource Association Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[12] =1) or (CPUID.(EAX=07H, ECX=0):EBX[15] =1))
		N-1:0	**Resource Monitoring ID (R/W):** ID for monitoring hardware to track internal operation, e.g. memory access.	N = Ceil (Log_2 (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		31:N	**Reserved**	
		63:32	**COS (R/W).** The class of service (COS) to enforce (on writes); returns the current COS when read.	If (CPUID.(EAX=07H, ECX=0):EBX.[15] = 1)
C90H - D8FH		Reserved MSR Address Space for CAT Mask Registers	See Section 17.19.4.1, "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.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C90H+n	3216+n	IA32_L3_MASK_n	**L3 CAT Mask for COSn (R/W)**	n = CPUID.(EAX=10H, ECX=1H):EDX[15:0]
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D10H - D4FH		Reserved MSR Address Space for L2 CAT Mask Registers	See Section 17.19.4.1, "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	
		1	**BNDPRESERVE:** Preserve the bounds registers for near branch instructions in the absence of the BND prefix	
		11:2	Reserved, must be 0	
		63:12	**Base Address of Bound Directory.**	
DA0H	3488	IA32_XSS	**Extended Supervisor State Mask (R/W)**	If(CPUID.(0DH, 1):EAX.[3] = 1
		7:0	**Reserved**	
		8	**Trace Packet Configuration State (R/W)**	
		63:9	Reserved.	
DB0H	3504	IA32_PKG_HDC_CTL	**Package Level Enable/disable HDC (R/W)**	If CPUID.06H:EAX.[13] = 1
		0	**HDC_Pkg_Enable (R/W)** Force HDC idling or wake up HDC-idled logical processors in the package. See Section 14.5.2, "Package level Enabling HDC"	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB1H	3505	IA32_PM_CTL1	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	**HDC_Allow_Block (R/W)** Allow/Block this logical processor for package level HDC control. See Section 14.5.3	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB2H	3506	IA32_THREAD_STALL	**Per-Logical_Processor HDC Idle Residency (R/0)**	If CPUID.06H:EAX.[13] = 1
		63:0	**Stall_Cycle_Cnt (R/W)** Stalled cycles due to HDC forced idle on this logical processor. See Section 14.5.4.1	If CPUID.06H:EAX.[13] = 1
4000_0000H - 4000_00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSR in this range.	
C000_0080H		IA32_EFER	**Extended Feature Enables**	If (CPUID.80000001H:EDX.[20] || CPUID.80000001H:EDX.[29])
		0	**SYSCALL Enable: IA32_EFER.SCE (R/W)** Enables SYSCALL/SYSRET instructions in 64-bit mode.	
		7:1	Reserved.	
		8	**IA-32e Mode Enable: IA32_EFER.LME (R/W)** Enables IA-32e mode operation.	
		9	Reserved.	
		10	**IA-32e Mode Active: IA32_EFER.LMA (R)** Indicates IA-32e mode is active when set.	
		11	**Execute Disable Bit Enable: IA32_EFER.NXE (R/W)**	
		63:12	Reserved.	
C000_0081H		IA32_STAR	**System Call Target Address (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	**IA-32e Mode System Call Target Address (R/W)** 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

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C000_0084H		IA32_FMASK	**System Call Flag Mask (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0100H		IA32_FS_BASE	**Map of BASE Address of FS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0101H		IA32_GS_BASE	**Map of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0102H		IA32_KERNEL_GS_BASE	**Swap Target of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0103H		IA32_TSC_AUX	Auxiliary TSC (RW)	If CPUID.80000001H: EDX[27] = 1
		31:0	AUX: Auxiliary signature of TSC	
		63:32	Reserved.	

NOTES:

1. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
2. The *_ADDR MSRs may or may not be present; this depends on flag settings in IA32_MC*i*_STATUS. See Section 15.3.2.3 and Section 15.3.2.4 for more information.
3. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have the CPUID signature DisplayFamily_DisplayModel of 06_17H.

The column "Shared/Unique" applies to multi-core processors based on Intel Core microarchitecture. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	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 processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		5		Reserved.
		6		Reserved.
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	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 64Processor (R/W)** See Table 2-2.
		3	Unique	**SMRR Enable (R/WL)** When this bit is set and the lock bit is set makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Unique	**Last Branch Record 0 From IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	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		▪ 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	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		▪ 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)
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
		11	Unique	**SMRR Capability Using MSR 0A0H and 0A1H (R)**
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	Shared/ Unique	Bit Description
Hex	Dec			
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	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.
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.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Shared	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19	Shared	**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Shared	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		36:35		Reserved.
		37	Unique	**DCU Prefetcher Disable (R/W)** When set to 1, The DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.
		38	Shared	**IDA Disable (R/W)** When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of IDA. If power-on default value is 1, IDA is available in the processor. If power-on default value is 0, IDA is not available.
		39	Unique	**IP Prefetcher Disable (R/W)** When set to 1, The IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.
		63:40		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
200H	512	IA32_MTRR_PHYSBASE0	Unique	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Unique	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Unique	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Unique	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Unique	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Unique	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Unique	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Unique	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Unique	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Unique	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Unique	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Unique	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Unique	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Unique	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Unique	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Unique	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Unique	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Unique	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Unique	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Unique	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Unique	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Unique	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Unique	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Unique	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Unique	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Unique	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Unique	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
309H	777	MSR_PERF_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)**

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30AH	778	IA32_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30AH	778	MSR_PERF_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)**
30BH	779	IA32_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
30BH	779	MSR_PERF_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)**
345H	837	IA32_PERF_CAPABILITIES	Unique	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
345H	837	MSR_PERF_CAPABILITIES	Unique	RO. This applies to processors that do not support architectural perfmon version 2.
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		63:8		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38DH	909	MSR_PERF_FIXED_CTR_ CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)**
38EH	910	IA32_PERF_GLOBAL_ STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATU S	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."
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_MC0_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC3_MISC	Unique	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	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).

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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 ADDRV 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 MISCV flag in the IA32_MCi_STATUS register is set.
419H	1045	IA32_MC6_STATUS	Unique	Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Chapter 23.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
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."
107CC H		MSR_EMON_L3_CTR_CTL0	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CD H		MSR_EMON_L3_CTR_CTL1	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CE H		MSR_EMON_L3_CTR_CTL2	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CF H		MSR_EMON_L3_CTR_CTL3	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H		MSR_EMON_L3_CTR_CTL4	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H		MSR_EMON_L3_CTR_CTL5	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H		MSR_EMON_L3_CTR_CTL6	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H		MSR_EMON_L3_CTR_CTL7	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
107D8H		MSR_EMON_L3_GL_CTL	Unique	**L3/FSB Common Control Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
C000_0080H		IA32_EFER	Unique	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

2.3 MSRS IN THE 45 NM AND 32 NM INTEL® ATOM™ PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1CH, 06_26H, 06_27H, 06_35H and 06_36H; see Table 2-1.

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

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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.
		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.**

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		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: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_ LASTBRANCH_4_FROM_IP	Unique	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_ LASTBRANCH_5_FROM_IP	Unique	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_ LASTBRANCH_6_FROM_IP	Unique	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_ LASTBRANCH_7_FROM_IP	Unique	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Unique	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Unique	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Unique	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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		▪ 111B: 083 MHz (FSB 333) ▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Shared	**Memory Type Range Register (R)** See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	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.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		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.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	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.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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.
		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.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		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		Reserved.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Unique	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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.
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.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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_MC0_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
40EH	1038	IA32_MC3_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
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.
C000_ 0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor with the CPUID signature with DisplayFamily_DisplayModel of 06_27H.

Table 2-5. MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C2_RESIDENCY	Package	**Package C2 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C2 Residency Counter. (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.

Table 2-5. MSRs Supported by Intel® Atom™ Processors (Contd.)with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
1BH	27	IA32_APIC_BASE	Core	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Module	**Processor Hard Power-On Configuration (R/W)** Writes ignored.
		63:0		Reserved **(R/O)**

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
		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.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	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
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Core	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**Reserved**
		22		**EN**
		23		**INV**
		31:24		**CMASK**

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:32		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.
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 degree C, The effective temperature for thermal throttling or PROCHOT# activation is “Temperature Target” + “Target Offset”
		29:24		**Target Offset (R/W)** Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).
		63:30		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Module	**Offcore Response Event Select Register (R/W)**
1A7H	423	MSR_OFFCORE_RSP_1	Module	**Offcore Response Event Select Register (R/W)**
1B0H	432	IA32_ENERGY_PERF_BIAS	Core	See Table 2-2.
1D9H	473	IA32_DEBUGCTL	Core	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Core	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Core	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.
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.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.
400H	1024	IA32_MC0_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Core	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Core	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Core	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Core	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Core	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	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_CTLS2	Core	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
491H	1169	IA32_VMX_FMFUNC	Core	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Core	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Core	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "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
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.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_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	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		7:0		Reserved.
		13:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		**See** Table 2-2
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Reserved**
		2		**Enable VMX outside SMX operation (R/WL)**
40H	64	MSR_LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5 and record format in Section 17.4.8.1
41H	65	MSR_LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only).
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
11EH	281	MSR_BBL_CR_CTL3	Module	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.
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.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
		38	Module	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
664H	1636	MSR_MC6_RESIDENCY_COU NTER	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.
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)** The is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.
		63:16		Reserved.

2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_37H) and Intel Atom processors (CPUID signatures with DisplayFamily_DisplayModel of 06_4AH, 06_5AH, 06_5DH).

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)**
		14:0		Package Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		23:17		Time Window for Power Limit #1. (R/W) in unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.
		63:24		Reserved
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8
CDH	205	MSR_FSB_FREQ	Module	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture.

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		▪ 100B: 080.0 MHz ▪ 000B: 083.3 MHz ▪ 001B: 100.0 MHz ▪ 010B: 133.3 MHz ▪ 011B: 116.7 MHz
		63:3		Reserved.

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	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.
		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	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**

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature

Register Address		Register Name	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.
		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 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
66EH	1646	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameter (R/O)**

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:0		Thermal Spec Power. (R/0) The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT
		63:15		Reserved

2.4.2 MSRs In Intel Atom Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID signature with DisplayFamily_DisplayModel including 06_4CH; see Table 2-1.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture

Address		Register Name	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:
		3:0		▪ 0000B: 083.3 MHz ▪ 0001B: 100.0 MHz ▪ 0010B: 133.3 MHz ▪ 0011B: 116.7 MHz ▪ 0100B: 080.0 MHz ▪ 0101B: 093.3 MHz ▪ 0110B: 090.0 MHz ▪ 0111B: 088.9 MHz ▪ 1000B: 087.5 MHz
		63:5		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7
		9:3		Reserved.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	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/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - Deep Power Down Technology is the max C-State 010b - C7 is the max C-State to include
		63:19		Reserved.
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)**
		14:0		PP0 Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains."
		16		Reserved

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:17		Time Window for Power Limit #1. (R/W) Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.
		63:24		Reserved

2.5 MSRS IN 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

Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		49:0		Reserved.
		52:50		See Table 2-2.
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		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)** The is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

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: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
17DH	381	MSR_SMM_MCA_CAP	Core	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
188H	392	IA32_PERFEVTSEL2	Core	See Table 2-2.
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.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		6:4		Reserved.
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Package	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
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.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:3		Reserved.
1AAH	426	MSR_MISC_PWR_MGMT	Package	Miscellaneous Power Management Control; various model specific features enumeration. See http://biosbits.org.
		0		**EIST Hardware Coordination Disable (R/W)** When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		21:1		Reserved.
		22		**Thermal Interrupt Coordination Enable (R/W)** If set, then thermal interrupt on one core is routed to all cores.
		63:23		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode by Core Groups (RW)** **Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically.** **For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.**
		7:0	Package	**Maximum Ratio Limit for Active cores in Group 0** Maximum turbo ratio limit when number of active cores is less or equal to Group 0 threshold.
		15:8	Package	**Maximum Ratio Limit for Active cores in Group 1** Maximum turbo ratio limit when number of active cores is less or equal to Group 1 threshold and greater than Group 0 threshold.
		23:16	Package	**Maximum Ratio Limit for Active cores in Group 2** Maximum turbo ratio limit when number of active cores is less or equal to Group 2 threshold and greater than Group 1 threshold.
		31:24	Package	**Maximum Ratio Limit for Active cores in Group 3** Maximum turbo ratio limit when number of active cores is less or equal to Group 3 threshold and greater than Group 2 threshold.
		39:32	Package	**Maximum Ratio Limit for Active cores in Group 4** Maximum turbo ratio limit when number of active cores is less or equal to Group 4 threshold and greater than Group 3 threshold.
		47:40	Package	**Maximum Ratio Limit for Active cores in Group 5** Maximum turbo ratio limit when number of active cores is less or equal to Group 5 threshold and greater than Group 4 threshold.
		55:48	Package	**Maximum Ratio Limit for Active cores in Group 6** Maximum turbo ratio limit when number of active cores is less or equal to Group 6 threshold and greater than Group 5 threshold.
		63:56	Package	**Maximum Ratio Limit for Active cores in Group 7** Maximum turbo ratio limit when number of active cores is less or equal to Group 7 threshold and greater than Group 6 threshold.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
1AEH	430	MSR_TURBO_GROUP_CORE CNT	Package	**Group Size of Active Cores for Turbo Mode Operation (RW)** **Writes of 0 threshold is ignored**
		7:0	Package	**Group 0 Core Count Threshold** Maximum number of active cores to operate under Group 0 Max Turbo Ratio limit.
		15:8	Package	**Group 1 Core Count Threshold** Maximum number of active cores to operate under Group 1 Max Turbo Ratio limit. Must be greater than Group 0 Core Count.
		23:16	Package	**Group 2 Core Count Threshold** Maximum number of active cores to operate under Group 2 Max Turbo Ratio limit. Must be greater than Group 1 Core Count.
		31:24	Package	**Group 3 Core Count Threshold** Maximum number of active cores to operate under Group 3 Max Turbo Ratio limit. Must be greater than Group 2 Core Count.
		39:32	Package	**Group 4 Core Count Threshold** Maximum number of active cores to operate under Group 4 Max Turbo Ratio limit. Must be greater than Group 3 Core Count.
		47:40	Package	**Group 5 Core Count Threshold** Maximum number of active cores to operate under Group 5 Max Turbo Ratio limit. Must be greater than Group 4 Core Count.
		55:48	Package	**Group 6 Core Count Threshold** Maximum number of active cores to operate under Group 6 Max Turbo Ratio limit. Must be greater than Group 5 Core Count.
		63:56	Package	**Group 7 Core Count Threshold** Maximum number of active cores to operate under Group 7 Max Turbo Ratio limit. Must be greater than Group 6 Core Count and not less than the total number of processor cores in the package. E.g. specify 255.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "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**
		9		**EN_CALL_STACK**
		63:10		Reserved.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
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**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**
		33		**Ovf_FixedCtr1**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		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."
		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**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55		**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.
		58		**Set 1 to cause LBR_Frz = 1**
		59		**Set 1 to cause CTR_Frz = 1**
		60		**Set 1 to cause ASCI = 1**
		61		**Set 1 to cause Ovf_Uncore**
		62		**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	914	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
406H	1030	IA32_MC1_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
4C3H	1219	IA32_A_PMC2	Core	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Core	See Table 2-2.
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Core	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:1		Reserved.
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Core	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Core	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Core	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Core	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ
		48:32		**PacketByteCnt**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Core	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Core	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Core	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Core	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Core	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in Watts) is in unit of, 1W/2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in Joules) is in unit of, 1Joule/ (2^ESU); where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.
		15:13		Reserved
		19:16		Time Unit. Time related information (in seconds) is in unit of, 1S/2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.
		63:20		Reserved
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 2-19 for supported time unit encodings.
		14:13		Reserved.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC_IRTL1	Package	**Package C6/C7S Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 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.
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.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)**
		14:0		**Thermal Spec Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		15		Reserved.
		30:16		**Minimum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		31		Reserved.
		46:32		**Maximum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		47		Reserved.
		54:48		**Maximum Time Window (R/W)** Specified by 2^Y * (1.0 + Z/4.0) * Time_Unit, where "Y" is the unsigned integer value represented. by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT
		63:55		Reserved.
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names,
		63:0		**Package C10 Residency Counter. (R/O)** Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.
639H	1593	MSR_PP0_ENERGY_STATU S	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATU S	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		3		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		8:4		Reserved.
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		11		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		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**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24:20		Reserved.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		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.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction** . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.6 and record format in Section 17.4.8.1
		0:47		**From Linear Address (R/W)**
		62:48		Signed extension of bits 47:0.
		63		Mispred
681H	1665	MSR_ LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_ LASTBRANCH_2_FROM_IP	Core	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_ LASTBRANCH_3_FROM_IP	Core	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_ LASTBRANCH_4_FROM_IP	Core	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_ LASTBRANCH_5_FROM_IP	Core	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_ LASTBRANCH_6_FROM_IP	Core	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Core	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Core	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Core	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Core	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
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.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
691H	1681	MSR_ LASTBRANCH_17_FROM_IP	Core	**Last Branch Record 17 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_ LASTBRANCH_18_FROM_IP	Core	**Last Branch Record 18 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_ LASTBRANCH_19_FROM_IP	Core	**Last Branch Record 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Core	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Core	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Core	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Core	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Core	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Core	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Core	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Core	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Core	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Core	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Core	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Core	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the **Destination instruction** and elapsed cycles from last LBR update. See also: ▪ Section 17.6
		0:47		**Target Linear Address (R/W)**
		63:48		Elapsed cycles from last update to the LBR.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Core	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Core	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Core	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Core	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Core	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Core	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Core	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Core	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Core	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Core	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Core	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Core	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Core	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Core	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Core	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D0H	1744	MSR_ LASTBRANCH_16_TO_IP	Core	**Last Branch Record 16 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D1H	1745	MSR_ LASTBRANCH_17_TO_IP	Core	**Last Branch Record 17 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_ LASTBRANCH_18_TO_IP	Core	**Last Branch Record 18 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_ LASTBRANCH_19_TO_IP	Core	**Last Branch Record 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Core	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Core	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Core	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Core	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Core	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Core	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Core	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Core	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Core	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Core	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Core	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Core	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Core	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Core	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Core	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Core	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Core	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Core	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Core	x2APIC Spurious Interrupt Vector register (R/W)
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)

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
81CH	2076	IA32_X2APIC_TMR4	Core	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Core	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Core	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Core	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Core	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Core	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Core	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Core	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Core	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Core	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Core	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Core	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Core	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Core	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Core	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Core	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Core	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Core	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Core	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Core	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Core	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Core	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Core	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Core	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Core	x2APIC Self IPI register (W/O)
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**

Table 2-12. MSRs in Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
D12H	3346	IA32_L2_QOS_MASK_2	Module	**L2 Class Of Service Mask - COS 2 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D13H	3347	IA32_L2_QOS_MASK_3	Package	**L2 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L2 ways for COS 3 enforcement**
		63:20		**Reserved**
D90H	3472	IA32_BNDCFGS	Core	See Table 2-2.
DA0H	3488	IA32_XSS	Core	See Table 2-2.
See Table 2-6, and Table 2-12 for MSR definitions applicable to processors with CPUID signature 06_5CH.				

2.6 MSRS IN 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 supercede 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

Address		Register Name	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.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
8CH	140	IA32_SGXLEPUBKEYHASH0	Core	See Table 2-2.
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	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. (R/W)
		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**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
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: ▪ 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* From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP; *i* = 1-31.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of the 3 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: ▪ 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* To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP; *i* = 1-31.
DC0H	3520	MSR_LASTBRANCH_INFO_ 0	Core	**Last Branch Record 0 Additional Information (R/W)** One of the 3 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: ▪ 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.
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.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
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.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
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 THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 2-14 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 2-15. 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 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Running count of SMI events since last RESET.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
79H	121	IA32_BIOS_ UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_ SIGN_ID	Thread	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Thread	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Thread	**Performance Counter Register** See Table 2-2.
C3H	195	IA32_PMC2	Thread	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Thread	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	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)** The is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDC-TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDC/TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 133.33MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		23:16		Reserved.
		24		**Interrupt filtering enable (R/W)** When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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
		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 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	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_PERFEVTSEL0	Thread	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Core	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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 degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15	Package	**TDP Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		30:16	Package	**TDC Limit (R/W)** TDC limit in 1/8 Amp granularity.
		31	Package	**TDC Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "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).

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Package	See Table 2-2.
281H	641	IA32_MC1_CTL2	Package	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Core	See Table 2-2.
285H	645	IA32_MC5_CTL2	Core	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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 (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
390H	912	MSR_PERF_GLOBAL_OVF_ CTRL	Thread	**(R/W)**
		61		**CLR_UNC_Ovf** Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
406H	1030	IA32_MC1_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
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."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O).** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
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: ▪ 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.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)
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)

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
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."

2.7.1 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 2-15. 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 2-15. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (not available to model 06_2EH)
		7:0		**Maximum Turbo Ratio Limit 1C (R/O)** Maximum Turbo mode ratio limit with 1 core active.
		15:8		**Maximum Turbo Ratio Limit 2C (R/O)** Maximum Turbo mode ratio limit with 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."

Table 2-15. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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."
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_ PERFEVTSEL0	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."

2.7.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 2-14 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-16. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2EH.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
394H	816	MSR_W_PMON_FIXED_CTR	Package	Uncore W-box perfmon fixed counter
395H	817	MSR_W_PMON_FIXED_ CTR_CTL	Package	Uncore U-box perfmon fixed counter control MSR
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
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 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	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.
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.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C21H	3105	MSR_B0_PMON_BOX_ STATUS	Package	Uncore B-box 0 perfmon local box status MSR.
C22H	3106	MSR_B0_PMON_BOX_OVF_ CTRL	Package	Uncore B-box 0 perfmon local box overflow control MSR.
C30H	3120	MSR_B0_PMON_EVNT_ SEL0	Package	Uncore B-box 0 perfmon event select MSR.
C31H	3121	MSR_B0_PMON_CTR0	Package	Uncore B-box 0 perfmon counter MSR.
C32H	3122	MSR_B0_PMON_EVNT_ SEL1	Package	Uncore B-box 0 perfmon event select MSR.
C33H	3123	MSR_B0_PMON_CTR1	Package	Uncore B-box 0 perfmon counter MSR.
C34H	3124	MSR_B0_PMON_EVNT_ SEL2	Package	Uncore B-box 0 perfmon event select MSR.
C35H	3125	MSR_B0_PMON_CTR2	Package	Uncore B-box 0 perfmon counter MSR.
C36H	3126	MSR_B0_PMON_EVNT_ SEL3	Package	Uncore B-box 0 perfmon event select MSR.
C37H	3127	MSR_B0_PMON_CTR3	Package	Uncore B-box 0 perfmon counter MSR.
C40H	3136	MSR_S0_PMON_BOX_CTRL	Package	Uncore S-box 0 perfmon local box control MSR.
C41H	3137	MSR_S0_PMON_BOX_ STATUS	Package	Uncore S-box 0 perfmon local box status MSR.
C42H	3138	MSR_S0_PMON_BOX_OVF_ CTRL	Package	Uncore S-box 0 perfmon local box overflow control MSR.
C50H	3152	MSR_S0_PMON_EVNT_ SEL0	Package	Uncore S-box 0 perfmon event select MSR.
C51H	3153	MSR_S0_PMON_CTR0	Package	Uncore S-box 0 perfmon counter MSR.
C52H	3154	MSR_S0_PMON_EVNT_ SEL1	Package	Uncore S-box 0 perfmon event select MSR.
C53H	3155	MSR_S0_PMON_CTR1	Package	Uncore S-box 0 perfmon counter MSR.
C54H	3156	MSR_S0_PMON_EVNT_ SEL2	Package	Uncore S-box 0 perfmon event select MSR.
C55H	3157	MSR_S0_PMON_CTR2	Package	Uncore S-box 0 perfmon counter MSR.
C56H	3158	MSR_S0_PMON_EVNT_ SEL3	Package	Uncore S-box 0 perfmon event select MSR.
C57H	3159	MSR_S0_PMON_CTR3	Package	Uncore S-box 0 perfmon counter MSR.
C60H	3168	MSR_B1_PMON_BOX_CTRL	Package	Uncore B-box 1 perfmon local box control MSR.
C61H	3169	MSR_B1_PMON_BOX_ STATUS	Package	Uncore B-box 1 perfmon local box status MSR.
C62H	3170	MSR_B1_PMON_BOX_OVF_ CTRL	Package	Uncore B-box 1 perfmon local box overflow control MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C70H	3184	MSR_B1_PMON_EVNT_SEL0	Package	Uncore B-box 1 perfmon event select MSR.
C71H	3185	MSR_B1_PMON_CTR0	Package	Uncore B-box 1 perfmon counter MSR.
C72H	3186	MSR_B1_PMON_EVNT_SEL1	Package	Uncore B-box 1 perfmon event select MSR.
C73H	3187	MSR_B1_PMON_CTR1	Package	Uncore B-box 1 perfmon counter MSR.
C74H	3188	MSR_B1_PMON_EVNT_SEL2	Package	Uncore B-box 1 perfmon event select MSR.
C75H	3189	MSR_B1_PMON_CTR2	Package	Uncore B-box 1 perfmon counter MSR.
C76H	3190	MSR_B1_PMON_EVNT_SEL3	Package	Uncore B-box 1vperfmon event select MSR.
C77H	3191	MSR_B1_PMON_CTR3	Package	Uncore B-box 1 perfmon counter MSR.
C80H	3120	MSR_W_PMON_BOX_CTRL	Package	Uncore W-box perfmon local box control MSR.
C81H	3121	MSR_W_PMON_BOX_STATUS	Package	Uncore W-box perfmon local box status MSR.
C82H	3122	MSR_W_PMON_BOX_OVF_CTRL	Package	Uncore W-box perfmon local box overflow control MSR.
C90H	3136	MSR_W_PMON_EVNT_SEL0	Package	Uncore W-box perfmon event select MSR.
C91H	3137	MSR_W_PMON_CTR0	Package	Uncore W-box perfmon counter MSR.
C92H	3138	MSR_W_PMON_EVNT_SEL1	Package	Uncore W-box perfmon event select MSR.
C93H	3139	MSR_W_PMON_CTR1	Package	Uncore W-box perfmon counter MSR.
C94H	3140	MSR_W_PMON_EVNT_SEL2	Package	Uncore W-box perfmon event select MSR.
C95H	3141	MSR_W_PMON_CTR2	Package	Uncore W-box perfmon counter MSR.
C96H	3142	MSR_W_PMON_EVNT_SEL3	Package	Uncore W-box perfmon event select MSR.
C97H	3143	MSR_W_PMON_CTR3	Package	Uncore W-box perfmon counter MSR.
CA0H	3232	MSR_M0_PMON_BOX_CTRL	Package	Uncore M-box 0 perfmon local box control MSR.
CA1H	3233	MSR_M0_PMON_BOX_STATUS	Package	Uncore M-box 0 perfmon local box status MSR.
CA2H	3234	MSR_M0_PMON_BOX_OVF_CTRL	Package	Uncore M-box 0 perfmon local box overflow control MSR.
CA4H	3236	MSR_M0_PMON_TIMESTAMP	Package	Uncore M-box 0 perfmon time stamp unit select MSR.
CA5H	3237	MSR_M0_PMON_DSP	Package	Uncore M-box 0 perfmon DSP unit select MSR.
CA6H	3238	MSR_M0_PMON_ISS	Package	Uncore M-box 0 perfmon ISS unit select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CA7H	3239	MSR_M0_PMON_MAP	Package	Uncore M-box 0 perfmon MAP unit select MSR.
CA8H	3240	MSR_M0_PMON_MSC_THR	Package	Uncore M-box 0 perfmon MIC THR select MSR.
CA9H	3241	MSR_M0_PMON_PGT	Package	Uncore M-box 0 perfmon PGT unit select MSR.
CAAH	3242	MSR_M0_PMON_PLD	Package	Uncore M-box 0 perfmon PLD unit select MSR.
CABH	3243	MSR_M0_PMON_ZDP	Package	Uncore M-box 0 perfmon ZDP unit select MSR.
CB0H	3248	MSR_M0_PMON_EVNT_SEL0	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTR0	Package	Uncore M-box 0 perfmon counter MSR.
CB2H	3250	MSR_M0_PMON_EVNT_SEL1	Package	Uncore M-box 0 perfmon event select MSR.
CB3H	3251	MSR_M0_PMON_CTR1	Package	Uncore M-box 0 perfmon counter MSR.
CB4H	3252	MSR_M0_PMON_EVNT_SEL2	Package	Uncore M-box 0 perfmon event select MSR.
CB5H	3253	MSR_M0_PMON_CTR2	Package	Uncore M-box 0 perfmon counter MSR.
CB6H	3254	MSR_M0_PMON_EVNT_SEL3	Package	Uncore M-box 0 perfmon event select MSR.
CB7H	3255	MSR_M0_PMON_CTR3	Package	Uncore M-box 0 perfmon counter MSR.
CB8H	3256	MSR_M0_PMON_EVNT_SEL4	Package	Uncore M-box 0 perfmon event select MSR.
CB9H	3257	MSR_M0_PMON_CTR4	Package	Uncore M-box 0 perfmon counter MSR.
CBAH	3258	MSR_M0_PMON_EVNT_SEL5	Package	Uncore M-box 0 perfmon event select MSR.
CBBH	3259	MSR_M0_PMON_CTR5	Package	Uncore M-box 0 perfmon counter MSR.
CC0H	3264	MSR_S1_PMON_BOX_CTRL	Package	Uncore S-box 1 perfmon local box control MSR.
CC1H	3265	MSR_S1_PMON_BOX_STATUS	Package	Uncore S-box 1 perfmon local box status MSR.
CC2H	3266	MSR_S1_PMON_BOX_OVF_CTRL	Package	Uncore S-box 1 perfmon local box overflow control MSR.
CD0H	3280	MSR_S1_PMON_EVNT_SEL0	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTR0	Package	Uncore S-box 1 perfmon counter MSR.
CD2H	3282	MSR_S1_PMON_EVNT_SEL1	Package	Uncore S-box 1 perfmon event select MSR.
CD3H	3283	MSR_S1_PMON_CTR1	Package	Uncore S-box 1 perfmon counter MSR.
CD4H	3284	MSR_S1_PMON_EVNT_SEL2	Package	Uncore S-box 1 perfmon event select MSR.
CD5H	3285	MSR_S1_PMON_CTR2	Package	Uncore S-box 1 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	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.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D01H	3329	MSR_C0_PMON_BOX_ STATUS	Package	Uncore C-box 0 perfmon local box status MSR.
D02H	3330	MSR_C0_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_C0_PMON_EVNT_ SEL0	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_C0_PMON_EVNT_ SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_C0_PMON_EVNT_ SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_C0_PMON_EVNT_ SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_C0_PMON_EVNT_ SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_C0_PMON_CTR4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_C0_PMON_EVNT_ SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_C0_PMON_CTR5	Package	Uncore C-box 0 perfmon counter MSR.
D20H	3360	MSR_C4_PMON_BOX_CTRL	Package	Uncore C-box 4 perfmon local box control MSR.
D21H	3361	MSR_C4_PMON_BOX_ STATUS	Package	Uncore C-box 4 perfmon local box status MSR.
D22H	3362	MSR_C4_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 4 perfmon local box overflow control MSR.
D30H	3376	MSR_C4_PMON_EVNT_ SEL0	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter MSR.
D32H	3378	MSR_C4_PMON_EVNT_ SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter MSR.
D34H	3380	MSR_C4_PMON_EVNT_ SEL2	Package	Uncore C-box 4 perfmon event select MSR.
D35H	3381	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter MSR.
D36H	3382	MSR_C4_PMON_EVNT_ SEL3	Package	Uncore C-box 4 perfmon event select MSR.
D37H	3383	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D38H	3384	MSR_C4_PMON_EVNT_SEL4	Package	Uncore C-box 4 perfmon event select MSR.
D39H	3385	MSR_C4_PMON_CTR4	Package	Uncore C-box 4 perfmon counter MSR.
D3AH	3386	MSR_C4_PMON_EVNT_SEL5	Package	Uncore C-box 4 perfmon event select MSR.
D3BH	3387	MSR_C4_PMON_CTR5	Package	Uncore C-box 4 perfmon counter MSR.
D40H	3392	MSR_C2_PMON_BOX_CTRL	Package	Uncore C-box 2 perfmon local box control MSR.
D41H	3393	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon local box status MSR.
D42H	3394	MSR_C2_PMON_BOX_OVF_CTRL	Package	Uncore C-box 2 perfmon local box overflow control MSR.
D50H	3408	MSR_C2_PMON_EVNT_SEL0	Package	Uncore C-box 2 perfmon event select MSR.
D51H	3409	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter MSR.
D52H	3410	MSR_C2_PMON_EVNT_SEL1	Package	Uncore C-box 2 perfmon event select MSR.
D53H	3411	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter MSR.
D54H	3412	MSR_C2_PMON_EVNT_SEL2	Package	Uncore C-box 2 perfmon event select MSR.
D55H	3413	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter MSR.
D56H	3414	MSR_C2_PMON_EVNT_SEL3	Package	Uncore C-box 2 perfmon event select MSR.
D57H	3415	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter MSR.
D58H	3416	MSR_C2_PMON_EVNT_SEL4	Package	Uncore C-box 2 perfmon event select MSR.
D59H	3417	MSR_C2_PMON_CTR4	Package	Uncore C-box 2 perfmon counter MSR.
D5AH	3418	MSR_C2_PMON_EVNT_SEL5	Package	Uncore C-box 2 perfmon event select MSR.
D5BH	3419	MSR_C2_PMON_CTR5	Package	Uncore C-box 2 perfmon counter MSR.
D60H	3424	MSR_C6_PMON_BOX_CTRL	Package	Uncore C-box 6 perfmon local box control MSR.
D61H	3425	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon local box status MSR.
D62H	3426	MSR_C6_PMON_BOX_OVF_CTRL	Package	Uncore C-box 6 perfmon local box overflow control MSR.
D70H	3440	MSR_C6_PMON_EVNT_SEL0	Package	Uncore C-box 6 perfmon event select MSR.
D71H	3441	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D72H	3442	MSR_C6_PMON_EVNT_SEL1	Package	Uncore C-box 6 perfmon event select MSR.
D73H	3443	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter MSR.
D74H	3444	MSR_C6_PMON_EVNT_SEL2	Package	Uncore C-box 6 perfmon event select MSR.
D75H	3445	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter MSR.
D76H	3446	MSR_C6_PMON_EVNT_SEL3	Package	Uncore C-box 6 perfmon event select MSR.
D77H	3447	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter MSR.
D78H	3448	MSR_C6_PMON_EVNT_SEL4	Package	Uncore C-box 6 perfmon event select MSR.
D79H	3449	MSR_C6_PMON_CTR4	Package	Uncore C-box 6 perfmon counter MSR.
D7AH	3450	MSR_C6_PMON_EVNT_SEL5	Package	Uncore C-box 6 perfmon event select MSR.
D7BH	3451	MSR_C6_PMON_CTR5	Package	Uncore C-box 6 perfmon counter MSR.
D80H	3456	MSR_C1_PMON_BOX_CTRL	Package	Uncore C-box 1 perfmon local box control MSR.
D81H	3457	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon local box status MSR.
D82H	3458	MSR_C1_PMON_BOX_OVF_CTRL	Package	Uncore C-box 1 perfmon local box overflow control MSR.
D90H	3472	MSR_C1_PMON_EVNT_SEL0	Package	Uncore C-box 1 perfmon event select MSR.
D91H	3473	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter MSR.
D92H	3474	MSR_C1_PMON_EVNT_SEL1	Package	Uncore C-box 1 perfmon event select MSR.
D93H	3475	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter MSR.
D94H	3476	MSR_C1_PMON_EVNT_SEL2	Package	Uncore C-box 1 perfmon event select MSR.
D95H	3477	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter MSR.
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.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DA0H	3488	MSR_C5_PMON_BOX_CTRL	Package	Uncore C-box 5 perfmon local box control MSR.
DA1H	3489	MSR_C5_PMON_BOX_ STATUS	Package	Uncore C-box 5 perfmon local box status MSR.
DA2H	3490	MSR_C5_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 5 perfmon local box overflow control MSR.
DB0H	3504	MSR_C5_PMON_EVNT_ SEL0	Package	Uncore C-box 5 perfmon event select MSR.
DB1H	3505	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter MSR.
DB2H	3506	MSR_C5_PMON_EVNT_ SEL1	Package	Uncore C-box 5 perfmon event select MSR.
DB3H	3507	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter MSR.
DB4H	3508	MSR_C5_PMON_EVNT_ SEL2	Package	Uncore C-box 5 perfmon event select MSR.
DB5H	3509	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter MSR.
DB6H	3510	MSR_C5_PMON_EVNT_ SEL3	Package	Uncore C-box 5 perfmon event select MSR.
DB7H	3511	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter MSR.
DB8H	3512	MSR_C5_PMON_EVNT_ SEL4	Package	Uncore C-box 5 perfmon event select MSR.
DB9H	3513	MSR_C5_PMON_CTR4	Package	Uncore C-box 5 perfmon counter MSR.
DBAH	3514	MSR_C5_PMON_EVNT_ SEL5	Package	Uncore C-box 5 perfmon event select MSR.
DBBH	3515	MSR_C5_PMON_CTR5	Package	Uncore C-box 5 perfmon counter MSR.
DC0H	3520	MSR_C3_PMON_BOX_CTRL	Package	Uncore C-box 3 perfmon local box control MSR.
DC1H	3521	MSR_C3_PMON_BOX_ STATUS	Package	Uncore C-box 3 perfmon local box status MSR.
DC2H	3522	MSR_C3_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 3 perfmon local box overflow control MSR.
DD0H	3536	MSR_C3_PMON_EVNT_ SEL0	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter MSR.
DD2H	3538	MSR_C3_PMON_EVNT_ SEL1	Package	Uncore C-box 3 perfmon event select MSR.
DD3H	3539	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter MSR.
DD4H	3540	MSR_C3_PMON_EVNT_ SEL2	Package	Uncore C-box 3 perfmon event select MSR.
DD5H	3541	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD6H	3542	MSR_C3_PMON_EVNT_SEL 3	Package	Uncore C-box 3 perfmon event select MSR.
DD7H	3543	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter MSR.
DD8H	3544	MSR_C3_PMON_EVNT_ SEL4	Package	Uncore C-box 3 perfmon event select MSR.
DD9H	3545	MSR_C3_PMON_CTR4	Package	Uncore C-box 3 perfmon counter MSR.
DDAH	3546	MSR_C3_PMON_EVNT_ SEL5	Package	Uncore C-box 3 perfmon event select MSR.
DDBH	3547	MSR_C3_PMON_CTR5	Package	Uncore C-box 3 perfmon counter MSR.
DE0H	3552	MSR_C7_PMON_BOX_CTRL	Package	Uncore C-box 7 perfmon local box control MSR.
DE1H	3553	MSR_C7_PMON_BOX_ STATUS	Package	Uncore C-box 7 perfmon local box status MSR.
DE2H	3554	MSR_C7_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 7 perfmon local box overflow control MSR.
DF0H	3568	MSR_C7_PMON_EVNT_ SEL0	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter MSR.
DF2H	3570	MSR_C7_PMON_EVNT_ SEL1	Package	Uncore C-box 7 perfmon event select MSR.
DF3H	3571	MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter MSR.
DF4H	3572	MSR_C7_PMON_EVNT_ SEL2	Package	Uncore C-box 7 perfmon event select MSR.
DF5H	3573	MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter MSR.
DF6H	3574	MSR_C7_PMON_EVNT_ SEL3	Package	Uncore C-box 7 perfmon event select MSR.
DF7H	3575	MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter MSR.
DF8H	3576	MSR_C7_PMON_EVNT_ SEL4	Package	Uncore C-box 7 perfmon event select MSR.
DF9H	3577	MSR_C7_PMON_CTR4	Package	Uncore C-box 7 perfmon counter MSR.
DFAH	3578	MSR_C7_PMON_EVNT_ SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTR5	Package	Uncore C-box 7 perfmon counter MSR.
E00H	3584	MSR_R0_PMON_BOX_CTRL	Package	Uncore R-box 0 perfmon local box control MSR.
E01H	3585	MSR_R0_PMON_BOX_ STATUS	Package	Uncore R-box 0 perfmon local box status MSR.
E02H	3586	MSR_R0_PMON_BOX_OVF_ CTRL	Package	Uncore R-box 0 perfmon local box overflow control MSR.
E04H	3588	MSR_R0_PMON_IPERF0_P0	Package	Uncore R-box 0 perfmon IPERF0 unit Port 0 select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E05H	3589	MSR_R0_PMON_IPERF0_P1	Package	Uncore R-box 0 perfmon IPERF0 unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERF0_P2	Package	Uncore R-box 0 perfmon IPERF0 unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERF0_P3	Package	Uncore R-box 0 perfmon IPERF0 unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERF0_P4	Package	Uncore R-box 0 perfmon IPERF0 unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERF0_P5	Package	Uncore R-box 0 perfmon IPERF0 unit Port 5 select MSR.
E0AH	3594	MSR_R0_PMON_IPERF0_P6	Package	Uncore R-box 0 perfmon IPERF0 unit Port 6 select MSR.
E0BH	3595	MSR_R0_PMON_IPERF0_P7	Package	Uncore R-box 0 perfmon IPERF0 unit Port 7 select MSR.
E0CH	3596	MSR_R0_PMON_QLX_P0	Package	Uncore R-box 0 perfmon QLX unit Port 0 select MSR.
E0DH	3597	MSR_R0_PMON_QLX_P1	Package	Uncore R-box 0 perfmon QLX unit Port 1 select MSR.
E0EH	3598	MSR_R0_PMON_QLX_P2	Package	Uncore R-box 0 perfmon QLX unit Port 2 select MSR.
E0FH	3599	MSR_R0_PMON_QLX_P3	Package	Uncore R-box 0 perfmon QLX unit Port 3 select MSR.
E10H	3600	MSR_R0_PMON_EVNT_SEL0	Package	Uncore R-box 0 perfmon event select MSR.
E11H	3601	MSR_R0_PMON_CTR0	Package	Uncore R-box 0 perfmon counter MSR.
E12H	3602	MSR_R0_PMON_EVNT_SEL1	Package	Uncore R-box 0 perfmon event select MSR.
E13H	3603	MSR_R0_PMON_CTR1	Package	Uncore R-box 0 perfmon counter MSR.
E14H	3604	MSR_R0_PMON_EVNT_SEL2	Package	Uncore R-box 0 perfmon event select MSR.
E15H	3605	MSR_R0_PMON_CTR2	Package	Uncore R-box 0 perfmon counter MSR.
E16H	3606	MSR_R0_PMON_EVNT_SEL3	Package	Uncore R-box 0 perfmon event select MSR.
E17H	3607	MSR_R0_PMON_CTR3	Package	Uncore R-box 0 perfmon counter MSR.
E18H	3608	MSR_R0_PMON_EVNT_SEL4	Package	Uncore R-box 0 perfmon event select MSR.
E19H	3609	MSR_R0_PMON_CTR4	Package	Uncore R-box 0 perfmon counter MSR.
E1AH	3610	MSR_R0_PMON_EVNT_SEL5	Package	Uncore R-box 0 perfmon event select MSR.
E1BH	3611	MSR_R0_PMON_CTR5	Package	Uncore R-box 0 perfmon counter MSR.
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.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E22H	3618	MSR_R1_PMON_BOX_OVF_CTRL	Package	Uncore R-box 1 perfmon local box overflow control MSR.
E24H	3620	MSR_R1_PMON_IPERF1_P8	Package	Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.
E25H	3621	MSR_R1_PMON_IPERF1_P9	Package	Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.
E26H	3622	MSR_R1_PMON_IPERF1_P10	Package	Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.
E27H	3623	MSR_R1_PMON_IPERF1_P11	Package	Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.
E28H	3624	MSR_R1_PMON_IPERF1_P12	Package	Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.
E29H	3625	MSR_R1_PMON_IPERF1_P13	Package	Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.
E2AH	3626	MSR_R1_PMON_IPERF1_P14	Package	Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.
E2BH	3627	MSR_R1_PMON_IPERF1_P15	Package	Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.
E2CH	3628	MSR_R1_PMON_QLX_P4	Package	Uncore R-box 1 perfmon QLX unit Port 4 select MSR.
E2DH	3629	MSR_R1_PMON_QLX_P5	Package	Uncore R-box 1 perfmon QLX unit Port 5 select MSR.
E2EH	3630	MSR_R1_PMON_QLX_P6	Package	Uncore R-box 1 perfmon QLX unit Port 6 select MSR.
E2FH	3631	MSR_R1_PMON_QLX_P7	Package	Uncore R-box 1 perfmon QLX unit Port 7 select MSR.
E30H	3632	MSR_R1_PMON_EVNT_SEL8	Package	Uncore R-box 1 perfmon event select MSR.
E31H	3633	MSR_R1_PMON_CTR8	Package	Uncore R-box 1 perfmon counter MSR.
E32H	3634	MSR_R1_PMON_EVNT_SEL9	Package	Uncore R-box 1 perfmon event select MSR.
E33H	3635	MSR_R1_PMON_CTR9	Package	Uncore R-box 1 perfmon counter MSR.
E34H	3636	MSR_R1_PMON_EVNT_SEL10	Package	Uncore R-box 1 perfmon event select MSR.
E35H	3637	MSR_R1_PMON_CTR10	Package	Uncore R-box 1 perfmon counter MSR.
E36H	3638	MSR_R1_PMON_EVNT_SEL11	Package	Uncore R-box 1 perfmon event select MSR.
E37H	3639	MSR_R1_PMON_CTR11	Package	Uncore R-box 1 perfmon counter MSR.
E38H	3640	MSR_R1_PMON_EVNT_SEL12	Package	Uncore R-box 1 perfmon event select MSR.
E39H	3641	MSR_R1_PMON_CTR12	Package	Uncore R-box 1 perfmon counter MSR.
E3AH	3642	MSR_R1_PMON_EVNT_SEL13	Package	Uncore R-box 1 perfmon event select MSR.
E3BH	3643	MSR_R1_PMON_CTR13	Package	Uncore R-box 1perfmon counter MSR.
E3CH	3644	MSR_R1_PMON_EVNT_SEL14	Package	Uncore R-box 1 perfmon event select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E3DH	3645	MSR_R1_PMON_CTR14	Package	Uncore R-box 1 perfmon counter MSR.
E3EH	3646	MSR_R1_PMON_EVNT_SEL15	Package	Uncore R-box 1 perfmon event select MSR.
E3FH	3647	MSR_R1_PMON_CTR15	Package	Uncore R-box 1 perfmon counter MSR.
E45H	3653	MSR_B0_PMON_MATCH	Package	Uncore B-box 0 perfmon local box match MSR.
E46H	3654	MSR_B0_PMON_MASK	Package	Uncore B-box 0 perfmon local box mask MSR.
E49H	3657	MSR_S0_PMON_MATCH	Package	Uncore S-box 0 perfmon local box match MSR.
E4AH	3658	MSR_S0_PMON_MASK	Package	Uncore S-box 0 perfmon local box mask MSR.
E4DH	3661	MSR_B1_PMON_MATCH	Package	Uncore B-box 1 perfmon local box match MSR.
E4EH	3662	MSR_B1_PMON_MASK	Package	Uncore B-box 1 perfmon local box mask MSR.
E54H	3668	MSR_M0_PMON_MM_CONFIG	Package	Uncore M-box 0 perfmon local box address match/mask config MSR.
E55H	3669	MSR_M0_PMON_ADDR_MATCH	Package	Uncore M-box 0 perfmon local box address match MSR.
E56H	3670	MSR_M0_PMON_ADDR_MASK	Package	Uncore M-box 0 perfmon local box address mask MSR.
E59H	3673	MSR_S1_PMON_MATCH	Package	Uncore S-box 1 perfmon local box match MSR.
E5AH	3674	MSR_S1_PMON_MASK	Package	Uncore S-box 1 perfmon local box mask MSR.
E5CH	3676	MSR_M1_PMON_MM_CONFIG	Package	Uncore M-box 1 perfmon local box address match/mask config MSR.
E5DH	3677	MSR_M1_PMON_ADDR_MATCH	Package	Uncore M-box 1 perfmon local box address match MSR.
E5EH	3678	MSR_M1_PMON_ADDR_MASK	Package	Uncore M-box 1 perfmon local box address mask MSR.
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

2.8 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 2-14, Table 2-15, plus additional MSR listed in Table 2-17. 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 2-17. Additional MSRs Supported by Intel Processors (Based on Intel® Microarchitecture Code Name Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		63:48		Reserved.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.

2.9 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 2-14 (except MSR address 1ADH), Table 2-15, plus additional MSR listed in Table 2-18. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2FH.

Table 2-18. Additional MSRs Supported by Intel® Xeon® Processor E7 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR.
F41H	3905	MSR_C8_PMON_BOX_ STATUS	Package	Uncore C-box 8 perfmon local box status MSR.
F42H	3906	MSR_C8_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR.
F50H	3920	MSR_C8_PMON_EVNT_ SEL0	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_ SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_ SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_ SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.
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 2-18. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)

Register Address		Register Name	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_ SEL0	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter MSR.
FD2H	4050	MSR_C9_PMON_EVNT_ SEL1	Package	Uncore C-box 9 perfmon event select MSR.
FD3H	4051	MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter MSR.
FD4H	4052	MSR_C9_PMON_EVNT_ SEL2	Package	Uncore C-box 9 perfmon event select MSR.
FD5H	4053	MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter MSR.
FD6H	4054	MSR_C9_PMON_EVNT_ SEL3	Package	Uncore C-box 9 perfmon event select MSR.
FD7H	4055	MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter MSR.
FD8H	4056	MSR_C9_PMON_EVNT_ SEL4	Package	Uncore C-box 9 perfmon event select MSR.
FD9H	4057	MSR_C9_PMON_CTR4	Package	Uncore C-box 9 perfmon counter MSR.
FDAH	4058	MSR_C9_PMON_EVNT_ SEL5	Package	Uncore C-box 9 perfmon event select MSR.
FDBH	4059	MSR_C9_PMON_CTR5	Package	Uncore C-box 9 perfmon counter MSR.

2.10 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 2-19 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 2-20.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, "MSRs in Pentium Processors."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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.
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)**
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	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_ PERFEVTSEL0	Thread	See Table 2-2.
187H	391	IA32_ PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_ PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_ PERFEVTSEL3	Thread	See Table 2-2.
18AH	394	IA32_ PERFEVTSEL4	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18BH	395	IA32_ PERFEVTSEL5	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18CH	396	IA32_ PERFEVTSEL6	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
18DH	397	IA32_ PERFEVTSEL7	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
198H	408	MSR_PERF_STATUS	Package	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/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		15:12		Reserved.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable** See Table 2-2
		6:1		Reserved.
		7	Thread	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Thread	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Thread	**xTPR Message Disable (R/W)** See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33:24		Reserved.
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Unique	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1A7H	422	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control; various model specific features enumeration. See http://biosbits.org.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B0H	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**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		63:15		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	See http://biosbits.org.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	Always 0 (CMCI not supported).
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		11:8		PEBS_REC_FORMAT. See Table 2-2.
		12		SMM_FREEZE. See Table 2-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_ STATUS		See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	**Ovf_PMC0**
		1	Thread	**Ovf_PMC1**
		2	Thread	**Ovf_PMC2**

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3	Thread	**Ovf_PMC3**
		4	Core	**Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Core	**Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		60:35		Reserved.
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "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)**
		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)**

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		62:36		Reserved.
		63		Enable Precise Store. (R/W)
3F6H	1014	MSR_PEBS_LD_LAT	Thread	see See Section 18.3.1.1.2, "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.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		**PCU Hardware Error (R/W)** When set, enables signaling of PCU hardware detected errors.
		1		**PCU Controller Error (R/W)** When set, enables signaling of PCU controller detected errors
		2		**PCU Firmware Error (R/W)** When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Thread	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
4C3H	1219	IA32_A_PMC2	Thread	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Thread	See Table 2-2.
4C5H	1221	IA32_A_PMC4	Core	See Table 2-2.
4C6H	1222	IA32_A_PMC5	Core	See Table 2-2.
4C7H	1223	IA32_A_PMC6	Core	See Table 2-2.
4C8H	1224	IA32_A_PMC7	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	**Package C6 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		**Package C2 Residency Counter. (R/O)** Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.9.1 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.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 2-2.
802H-83FH		X2APIC MSRs	Thread	See Table 2-2.
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature (R/W)** See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.10.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-20 and Table 2-21 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 2-20. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
60CH	1548	MSR_PKGC7_IRTL	Package	**Package C7 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-20. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
63AH	1594	MSR_PP0_POLICY	Package	**PP0 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-20, and Table 2-21 for MSR definitions applicable to processors with CPUID signature 06_2AH.				

Table 2-21 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06_2AH.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4 select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_ CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_ PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
702H	1794	MSR_UNC_CBO_0_ PERFEVTSEL2	Package	Uncore C-Box 0, counter 2 event select MSR.
703H	1795	MSR_UNC_CBO_0_ PERFEVTSEL3	Package	Uncore C-Box 0, counter 3 event select MSR.
705H	1797	MSR_UNC_CBO_0_UNIT_ STATUS	Package	Uncore C-Box 0, unit status for counter 0-3
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
708H	1800	MSR_UNC_CBO_0_PERFCTR2	Package	Uncore C-Box 0, performance counter 2.
709H	1801	MSR_UNC_CBO_0_PERFCTR3	Package	Uncore C-Box 0, performance counter 3.
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
712H	1810	MSR_UNC_CBO_1_ PERFEVTSEL2	Package	Uncore C-Box 1, counter 2 event select MSR.
713H	1811	MSR_UNC_CBO_1_ PERFEVTSEL3	Package	Uncore C-Box 1, counter 3 event select MSR.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
715H	1813	MSR_UNC_CBO_1_UNIT_ STATUS	Package	Uncore C-Box 1, unit status for counter 0-3
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
718H	1816	MSR_UNC_CBO_1_PERFCTR2	Package	Uncore C-Box 1, performance counter 2.
719H	1817	MSR_UNC_CBO_1_PERFCTR3	Package	Uncore C-Box 1, performance counter 3.
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
722H	1826	MSR_UNC_CBO_2_ PERFEVTSEL2	Package	Uncore C-Box 2, counter 2 event select MSR.
723H	1827	MSR_UNC_CBO_2_ PERFEVTSEL3	Package	Uncore C-Box 2, counter 3 event select MSR.
725H	1829	MSR_UNC_CBO_2_UNIT_ STATUS	Package	Uncore C-Box 2, unit status for counter 0-3
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
728H	1832	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
729H	1833	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
732H	1842	MSR_UNC_CBO_3_ PERFEVTSEL2	Package	Uncore C-Box 3, counter 2 event select MSR.
733H	1843	MSR_UNC_CBO_3_ PERFEVTSEL3	Package	Uncore C-Box 3, counter 3 event select MSR.
735H	1845	MSR_UNC_CBO_3_UNIT_ STATUS	Package	Uncore C-Box 3, unit status for counter 0-3
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
738H	1848	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
739H	1849	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
740H	1856	MSR_UNC_CBO_4_ PERFEVTSEL0	Package	Uncore C-Box 4, counter 0 event select MSR
741H	1857	MSR_UNC_CBO_4_ PERFEVTSEL1	Package	Uncore C-Box 4, counter 1 event select MSR.
742H	1858	MSR_UNC_CBO_4_ PERFEVTSEL2	Package	Uncore C-Box 4, counter 2 event select MSR.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
743H	1859	MSR_UNC_CBO_4_ PERFEVTSEL3	Package	Uncore C-Box 4, counter 3 event select MSR.
745H	1861	MSR_UNC_CBO_4_UNIT_ STATUS	Package	Uncore C-Box 4, unit status for counter 0-3
746H	1862	MSR_UNC_CBO_4_PERFCTR0	Package	Uncore C-Box 4, performance counter 0.
747H	1863	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, performance counter 1.
748H	1864	MSR_UNC_CBO_4_PERFCTR2	Package	Uncore C-Box 4, performance counter 2.
749H	1865	MSR_UNC_CBO_4_PERFCTR3	Package	Uncore C-Box 4, performance counter 3.

2.10.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-22 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 2-19 and Table 2-23.

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
39CH	924	MSR_PEBS_NUM_ALT	Package	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 19-17
		63:1		Reserved (must be zero).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATU S	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-22, and Table 2-23 for MSR definitions applicable to processors with CPUID signature 06_2DH.				

2.10.3 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 2-23. 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 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C08H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
C09H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C10H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
C11H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
C16H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
C17H		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
C24H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
C30H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
C31H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
C32H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
C33H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
C34H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
C36H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
C37H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
C38H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
C39H		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
D04H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon local box wide control.
D10H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
D11H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
D12H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
D13H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
D14H		MSR_C0_PMON_BOX_FILTER	Package	Uncore C-box 0 perfmon box wide filter.
D16H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
D17H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
D18H		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
D19H		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
D24H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon local box wide control.
D30H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
D31H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
D32H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
D33H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
D34H		MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 perfmon box wide filter.
D36H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
D37H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
D38H		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
D39H		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
D44H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon local box wide control.
D50H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
D51H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D52H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
D53H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
D54H		MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 perfmon box wide filter.
D56H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
D57H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
D58H		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
D59H		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
D64H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon local box wide control.
D70H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
D71H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
D72H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
D73H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
D74H		MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 perfmon box wide filter.
D76H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
D77H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
D78H		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
D79H		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
D84H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon local box wide control.
D90H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
D91H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
D92H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
D93H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
D94H		MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 perfmon box wide filter.
D96H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
D97H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
D98H		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
D99H		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
DA4H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon local box wide control.
DB0H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
DB1H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
DB2H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
DB3H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
DB4H		MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 perfmon box wide filter.
DB6H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
DB7H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
DB8H		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
DB9H		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC4H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon local box wide control.
DD0H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
DD1H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
DD2H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
DD3H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
DD4H		MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 perfmon box wide filter.
DD6H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
DD7H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
DD8H		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
DD9H		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
DE4H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon local box wide control.
DF0H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
DF1H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
DF2H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
DF3H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
DF4H		MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 perfmon box wide filter.
DF6H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
DF7H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
DF8H		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
DF9H		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.

2.11 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 2-19, Table 2-20, Table 2-21, and Table 2-24. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3AH.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
See Table 2-19, Table 2-20 and Table 2-21 for other MSR definitions applicable to processors with CPUID signature 06_3AH				

2.11.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 2-25 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 2-19, and Table 2-25.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		**Protected Processor Inventory Number (R/O)** A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
CEH	206	MSR_PLATFORM_INFO	Package	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)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for privileged system inventory agent to read PPIN from MSR_PPIN. When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		Reserved.
		26		**MCG_ELOG_P**
		63:27		Reserved.
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		27:24		**TCC Activation Offset (R/W)** Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT 1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		63:32		Reserved
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
296H	662	IA32_MC22_CTL2	Package	See Table 2-2.
297H	663	IA32_MC23_CTL2	Package	See Table 2-2.
298H	664	IA32_MC24_CTL2	Package	See Table 2-2.
299H	665	IA32_MC25_CTL2	Package	See Table 2-2.
29AH	666	IA32_MC26_CTL2	Package	See Table 2-2.
29BH	667	IA32_MC27_CTL2	Package	See Table 2-2.
29CH	668	IA32_MC28_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC7 and MC 8 report MC error from the two home agents.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC7 and MC 8 report MC error from the two home agents.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
42DH	1069	IA32_MC11_STATUS	Package	Bank MC11 reports MC error from a specific channel of the integrated memory controller.
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	Bank MC20 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
458H	1112	IA32_MC22_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC22 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
459H	1113	IA32_MC22_STATUS	Package	
45AH	1114	IA32_MC22_ADDR	Package	
45BH	1115	IA32_MC22_MISC	Package	
45CH	1116	IA32_MC23_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC23 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
45DH	1117	IA32_MC23_STATUS	Package	
45EH	1118	IA32_MC23_ADDR	Package	
45FH	1119	IA32_MC23_MISC	Package	
460H	1120	IA32_MC24_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC24 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
461H	1121	IA32_MC24_STATUS	Package	
462H	1122	IA32_MC24_ADDR	Package	
463H	1123	IA32_MC24_MISC	Package	
464H	1124	IA32_MC25_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC25 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
465H	1125	IA32_MC25_STATUS	Package	
466H	1126	IA32_MC25_ADDR	Package	
467H	1127	IA32_MC2MISC	Package	
468H	1128	IA32_MC26_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC26 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
469H	1129	IA32_MC26_STATUS	Package	
46AH	1130	IA32_MC26_ADDR	Package	
46BH	1131	IA32_MC26_MISC	Package	
46CH	1132	IA32_MC27_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC27 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
46DH	1133	IA32_MC27_STATUS	Package	
46EH	1134	IA32_MC27_ADDR	Package	
46FH	1135	IA32_MC27_MISC	Package	

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
470H	1136	IA32_MC28_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC28 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
471H	1137	IA32_MC28_STATUS	Package	
472H	1138	IA32_MC28_ADDR	Package	
473H	1139	IA32_MC28_MISC	Package	
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06_3EH				

2.11.2 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 2-19, Table 2-25, and Table 2-26.

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		63:25		Reserved.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status (R/WO)
		0		**RIPV**
		1		**EIPV**
		2		**MCIP**
		3		**LMCE signaled**
		63:4		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		62:56		Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
29DH	669	IA32_MC29_CTL2	Package	See Table 2-2.
29EH	670	IA32_MC30_CTL2	Package	See Table 2-2.
29FH	671	IA32_MC31_CTL2	Package	See Table 2-2.

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "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.
41BH	1051	IA32_MC6_MISC	Package	Misc MAC information of Integrated I/O. (R/O) see Section 15.3.2.4
		5:0		Recoverable Address LSB
		8:6		Address Mode
		15:9		Reserved
		31:16		PCI Express Requestor ID
		39:32		PCI Express Segment Number
		63:32		Reserved
474H	1140	IA32_MC29_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC29 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
475H	1141	IA32_MC29_STATUS	Package	
476H	1142	IA32_MC29_ADDR	Package	
477H	1143	IA32_MC29_MISC	Package	
478H	1144	IA32_MC30_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC30 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
479H	1145	IA32_MC30_STATUS	Package	
47AH	1146	IA32_MC30_ADDR	Package	
47BH	1147	IA32_MC30_MISC	Package	
47CH	1148	IA32_MC31_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC31 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
47DH	1149	IA32_MC31_STATUS	Package	
47EH	1150	IA32_MC31_ADDR	Package	
47FH	1147	IA32_MC31_MISC	Package	
See Table 2-19, Table 2-25 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.

2.11.3 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 2-23 and Table 2-27. 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 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C00H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
C01H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
C06H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
C15H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
C35H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
D1AH		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter1.
D3AH		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
D5AH		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
D7AH		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
D9AH		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
DBAH		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
DDAH		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
DFAH		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E04H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E10H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E11H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E12H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E13H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E14H		MSR_C8_PMON_BOX_FILTER	Package	Uncore C-box 8 perfmon box wide filter.
E16H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E17H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E18H		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E19H		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E1AH		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E24H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E30H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E31H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E32H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E33H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E34H		MSR_C9_PMON_BOX_FILTER	Package	Uncore C-box 9 perfmon box wide filter.
E36H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E37H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.

Table 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E38H		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E39H		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
E3AH		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E44H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
E50H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
E51H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
E52H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
E53H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
E54H		MSR_C10_PMON_BOX_FILTER	Package	Uncore C-box 10 perfmon box wide filter.
E56H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.
E57H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
E58H		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
E59H		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
E5AH		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
E64H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
E70H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
E71H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
E72H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
E73H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
E74H		MSR_C11_PMON_BOX_FILTER	Package	Uncore C-box 11 perfmon box wide filter.
E76H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
E77H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
E78H		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
E79H		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
E7AH		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
E84H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
E90H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
E91H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
E92H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
E93H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
E94H		MSR_C12_PMON_BOX_FILTER	Package	Uncore C-box 12 perfmon box wide filter.
E96H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
E97H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
E98H		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
E99H		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
E9AH		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EA4H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.

Table 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB0H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
EB1H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
EB2H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
EB3H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
EB4H		MSR_C13_PMON_BOX_FILTER	Package	Uncore C-box 13 perfmon box wide filter.
EB6H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
EB7H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EB8H		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EB9H		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EBAH		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
EC4H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
ED0H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
ED1H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
ED2H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
ED3H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
ED4H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter.
ED6H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
ED7H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
ED8H		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
ED9H		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EDAH		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.

2.12 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 2-19, Table 2-20, Table 2-21, and Table 2-28. For an MSR listed in Table 2-19 that also appears in Table 2-28, Table 2-28 supercede Table 2-19.

The MSRs listed in Table 2-28 also apply to processors based on Haswell-E microarchitecture (see Section 2.13).

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
186H	390	IA32_PERFEVTSEL0	THREAD	**Performance Event Select for Counter 0 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.3.6.5.1 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 inTable 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 inTable 2-2 and the fields below.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		32		IN_TX: see Section 18.3.6.5.1 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, spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large "sample-after" value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.
189H	393	IA32_PERFEVTSEL3	THREAD	**Performance Event Select for Counter 3 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.3.6.5.1 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**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		9		**EN_CALL_STACK**
		63:9		Reserved.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
		0		**LBR: Last Branch Record**
		1		**BTF**
		5:2		Reserved.
		6		**TR: Branch Trace**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		15		**RTM_DEBUG**
		63:15		Reserved.
491H	1169	IA32_VMX_VMFUNC	THREAD	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
60BH	1548	MSR_PKGC_IRTL1	Package	**Package C6/C7 Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C6/C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 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 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		62:47		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		62:47		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
C80H	3200	IA32_DEBUG_INTERFACE	Package	**Silicon Debug Feature Control (R/W)** See Table 2-2.

2.12.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 2-29 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 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processor with signature 06_3CH
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Encoded number of C-Box, derive value by "-1"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_ PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
391H	913	MSR_UNC_PERF_GLOBAL_ CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
395H	917	MSR_UNC_PERF_FIXED_ CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
4E0H	1248	MSR_SMM_FEATURE_CONTR OL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W)** **(frequency refers to processor graphics frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Graphics Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:30		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W)** **(frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		5:2		Reserved.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1824	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
See Table 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06_46H.				

2.12.2 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 2-19, Table 2-20, Table 2-28, Table 2-29, and Table 2-30.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		59:0		Package C10 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved
See Table 2-19, Table 2-20, Table 2-21, Table 2-28, Table 2-29 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

2.13 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 2-19, Table 2-28, and Table 2-31.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	**Configured State of Enabled Processor Core Count and Logical Processor Count (RO)** ▪ After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package. ▪ Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.
		15:0		**Core_COUNT (RO)** The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		31:16		**THREAD_COUNT (RO)** The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		63:32		Reserved
53H	83	MSR_THREAD_ID_INFO	Thread	**A Hardware Assigned ID for the Logical Processor (RO)**
		7:0		**Logical_Processor_ID (RO)** An implementation-specific numerical. value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.
		63:8		Reserved

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

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.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10 core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		63:56	Package	**Maximum Ratio Limit for16C** Maximum turbo ratio limit of 16 core active.
1AFH	431	MSR_TURBO_RATIO_LIMIT2	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 17C** Maximum turbo ratio limit of 17 core active.
		15:8	Package	**Maximum Ratio Limit for 18C** Maximum turbo ratio limit of 18 core active.
		62:16	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** Energy Consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
61EH	1566	MSR_PCIE_PLL_RATIO	Package	**Configuration of PCIE PLL Relative to BCLK(R/W)**
		1:0	Package	**PCIE Ratio (R/W)** 00b: Use 5:5 mapping for100MHz operation (default) 01b: Use 5:4 mapping for125MHz operation 10b: Use 5:3 mapping for166MHz operation 11b: Use 5:2 mapping for250MHz operation
		2	Package	**LPLL Select (R/W)** if 1, use configured setting of PCIE Ratio
		3	Package	**LONG RESET (R/W)** if 1, wait additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved
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_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_RE ASONS	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.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		**Multi-Core Turbo Log** When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		**Core Frequency P1 Log** When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Core Max n-core Turbo Frequency Limiting Log** When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		**Core Frequency Limiting Log** When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x0: no monitoring** **0x1: L3 occupancy monitoring** **all other encoding reserved.**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8EH	3214	IA32_QM_CTR	THREAD	**Monitoring Counter Register (R/O).** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		61:0		**Resource Monitored Data**
		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 2-19, Table 2-28 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.

2.13.1 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 2-32. 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 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
700H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
701H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
702H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
703H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
704H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
705H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
706H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
708H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
709H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
70AH		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
710H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
711H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
712H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
713H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
714H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
715H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
716H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
717H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
718H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
719H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
71AH		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
720H		MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 perfmon for SBo 0 box-wide control
721H		MSR_S0_PMON_EVNTSEL0	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 0.
722H		MSR_S0_PMON_EVNTSEL1	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 1.
723H		MSR_S0_PMON_EVNTSEL2	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 2.
724H		MSR_S0_PMON_EVNTSEL3	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 3.
725H		MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 perfmon box-wide filter.
726H		MSR_S0_PMON_CTR0	Package	Uncore SBo 0 perfmon counter 0.
727H		MSR_S0_PMON_CTR1	Package	Uncore SBo 0 perfmon counter 1.
728H		MSR_S0_PMON_CTR2	Package	Uncore SBo 0 perfmon counter 2.
729H		MSR_S0_PMON_CTR3	Package	Uncore SBo 0 perfmon counter 3.
72AH		MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 perfmon for SBo 1 box-wide control
72BH		MSR_S1_PMON_EVNTSEL0	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 0.
72CH		MSR_S1_PMON_EVNTSEL1	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 1.
72DH		MSR_S1_PMON_EVNTSEL2	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 2.
72EH		MSR_S1_PMON_EVNTSEL3	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 3.
72FH		MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 perfmon box-wide filter.
730H		MSR_S1_PMON_CTR0	Package	Uncore SBo 1 perfmon counter 0.
731H		MSR_S1_PMON_CTR1	Package	Uncore SBo 1 perfmon counter 1.
732H		MSR_S1_PMON_CTR2	Package	Uncore SBo 1 perfmon counter 2.
733H		MSR_S1_PMON_CTR3	Package	Uncore SBo 1 perfmon counter 3.
734H		MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 perfmon for SBo 2 box-wide control

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
735H		MSR_S2_PMON_EVNTSEL0	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 0.
736H		MSR_S2_PMON_EVNTSEL1	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 1.
737H		MSR_S2_PMON_EVNTSEL2	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 2.
738H		MSR_S2_PMON_EVNTSEL3	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 3.
739H		MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 perfmon box-wide filter.
73AH		MSR_S2_PMON_CTR0	Package	Uncore SBo 2 perfmon counter 0.
73BH		MSR_S2_PMON_CTR1	Package	Uncore SBo 2 perfmon counter 1.
73CH		MSR_S2_PMON_CTR2	Package	Uncore SBo 2 perfmon counter 2.
73DH		MSR_S2_PMON_CTR3	Package	Uncore SBo 2 perfmon counter 3.
73EH		MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 perfmon for SBo 3 box-wide control
73FH		MSR_S3_PMON_EVNTSEL0	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 0.
740H		MSR_S3_PMON_EVNTSEL1	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 1.
741H		MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 2.
742H		MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 3.
743H		MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 perfmon box-wide filter.
744H		MSR_S3_PMON_CTR0	Package	Uncore SBo 3 perfmon counter 0.
745H		MSR_S3_PMON_CTR1	Package	Uncore SBo 3 perfmon counter 1.
746H		MSR_S3_PMON_CTR2	Package	Uncore SBo 3 perfmon counter 2.
747H		MSR_S3_PMON_CTR3	Package	Uncore SBo 3 perfmon counter 3.
E00H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon for box-wide control
E01H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
E02H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
E03H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
E04H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
E05H		MSR_C0_PMON_BOX_FILTER0	Package	Uncore C-box 0 perfmon box wide filter 0.
E06H		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter 1.
E07H		MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon box wide status.
E08H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
E09H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
E0AH		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
E0BH		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
E10H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon for box-wide control
E11H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
E12H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
E13H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
E14H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
E15H		MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-box 1 perfmon box wide filter 0.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E16H		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
E17H		MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon box wide status.
E18H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
E19H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
E1AH		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
E1BH		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
E20H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon for box-wide control
E21H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
E22H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.
E23H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
E24H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
E25H		MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-box 2 perfmon box wide filter 0.
E26H		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
E27H		MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon box wide status.
E28H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
E29H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
E2AH		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
E2BH		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
E30H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon for box-wide control
E31H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
E32H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
E33H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
E34H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
E35H		MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-box 3 perfmon box wide filter 0.
E36H		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
E37H		MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon box wide status.
E38H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
E39H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
E3AH		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
E3BH		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
E40H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon for box-wide control
E41H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
E42H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
E43H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
E44H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
E45H		MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-box 4 perfmon box wide filter 0.
E46H		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E47H		MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon box wide status.
E48H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
E49H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
E4AH		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
E4BH		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
E50H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon for box-wide control
E51H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
E52H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
E53H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
E54H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
E55H		MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-box 5 perfmon box wide filter 0.
E56H		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
E57H		MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon box wide status.
E58H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
E59H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
E5AH		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
E5BH		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.
E60H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon for box-wide control
E61H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
E62H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
E63H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
E64H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
E65H		MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-box 6 perfmon box wide filter 0.
E66H		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
E67H		MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon box wide status.
E68H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
E69H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
E6AH		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
E6BH		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
E70H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon for box-wide control.
E71H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
E72H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
E73H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
E74H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
E75H		MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-box 7 perfmon box wide filter 0.
E76H		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E77H		MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon box wide status.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E78H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
E79H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
E7AH		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
E7BH		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.
E80H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E81H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E82H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E83H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E84H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E85H		MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-box 8 perfmon box wide filter0.
E86H		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E87H		MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon box wide status.
E88H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E89H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E8AH		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E8BH		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E90H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E91H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E92H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E93H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E94H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E95H		MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-box 9 perfmon box wide filter0.
E96H		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E97H		MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon box wide status.
E98H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E99H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.
E9AH		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E9BH		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
EA0H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
EA1H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
EA2H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
EA3H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
EA4H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
EA5H		MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-box 10 perfmon box wide filter0.
EA6H		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
EA7H		MSR_C10_PMON_BOX_STATUS	Package	Uncore C-box 10 perfmon box wide status.
EA8H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EA9H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
EAAH		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
EABH		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
EB0H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
EB1H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
EB2H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
EB3H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
EB4H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
EB5H		MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-box 11 perfmon box wide filter0.
EB6H		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
EB7H		MSR_C11_PMON_BOX_STATUS	Package	Uncore C-box 11 perfmon box wide status.
EB8H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
EB9H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
EBAH		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
EBBH		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
EC0H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
EC1H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
EC2H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
EC3H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
EC4H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
EC5H		MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-box 12 perfmon box wide filter0.
EC6H		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EC7H		MSR_C12_PMON_BOX_STATUS	Package	Uncore C-box 12 perfmon box wide status.
EC8H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
EC9H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
ECAH		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
ECBH		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
ED0H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.
ED1H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
ED2H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
ED3H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
ED4H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
ED5H		MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-box 13 perfmon box wide filter0.
ED6H		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
ED7H		MSR_C13_PMON_BOX_STATUS	Package	Uncore C-box 13 perfmon box wide status.
ED8H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
ED9H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EDAH		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EDBH		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EE0H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
EE1H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
EE2H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
EE3H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
EE4H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
EE5H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter0.
EE6H		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.
EE7H		MSR_C14_PMON_BOX_STATUS	Package	Uncore C-box 14 perfmon box wide status.
EE8H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
EE9H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
EEAH		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
EEBH		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EF0H		MSR_C15_PMON_BOX_CTL	Package	Uncore C-box 15 perfmon local box wide control.
EF1H		MSR_C15_PMON_EVNTSEL0	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 0.
EF2H		MSR_C15_PMON_EVNTSEL1	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 1.
EF3H		MSR_C15_PMON_EVNTSEL2	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 2.
EF4H		MSR_C15_PMON_EVNTSEL3	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 3.
EF5H		MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-box 15 perfmon box wide filter0.
EF6H		MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-box 15 perfmon box wide filter1.
EF7H		MSR_C15_PMON_BOX_STATUS	Package	Uncore C-box 15 perfmon box wide status.
EF8H		MSR_C15_PMON_CTR0	Package	Uncore C-box 15 perfmon counter 0.
EF9H		MSR_C15_PMON_CTR1	Package	Uncore C-box 15 perfmon counter 1.
EFAH		MSR_C15_PMON_CTR2	Package	Uncore C-box 15 perfmon counter 2.
EFBH		MSR_C15_PMON_CTR3	Package	Uncore C-box 15 perfmon counter 3.
F00H		MSR_C16_PMON_BOX_CTL	Package	Uncore C-box 16 perfmon for box-wide control
F01H		MSR_C16_PMON_EVNTSEL0	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 0.
F02H		MSR_C16_PMON_EVNTSEL1	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 1.
F03H		MSR_C16_PMON_EVNTSEL2	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 2.
F04H		MSR_C16_PMON_EVNTSEL3	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 3.
F05H		MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-box 16 perfmon box wide filter 0.
F06H		MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-box 16 perfmon box wide filter 1.
F07H		MSR_C16_PMON_BOX_STATUS	Package	Uncore C-box 16 perfmon box wide status.
F08H		MSR_C16_PMON_CTR0	Package	Uncore C-box 16 perfmon counter 0.
F09H		MSR_C16_PMON_CTR1	Package	Uncore C-box 16 perfmon counter 1.
F0AH		MSR_C16_PMON_CTR2	Package	Uncore C-box 16 perfmon counter 2.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E0BH		MSR_C16_PMON_CTR3	Package	Uncore C-box 16 perfmon counter 3.
F10H		MSR_C17_PMON_BOX_CTL	Package	Uncore C-box 17 perfmon for box-wide control
F11H		MSR_C17_PMON_EVNTSEL0	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 0.
F12H		MSR_C17_PMON_EVNTSEL1	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 1.
F13H		MSR_C17_PMON_EVNTSEL2	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 2.
F14H		MSR_C17_PMON_EVNTSEL3	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 3.
F15H		MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-box 17 perfmon box wide filter 0.
F16H		MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-box 17 perfmon box wide filter1.
F17H		MSR_C17_PMON_BOX_STATUS	Package	Uncore C-box 17 perfmon box wide status.
F18H		MSR_C17_PMON_CTR0	Package	Uncore C-box 17 perfmon counter 0.
F19H		MSR_C17_PMON_CTR1	Package	Uncore C-box 17 perfmon counter 1.
F1AH		MSR_C17_PMON_CTR2	Package	Uncore C-box 17 perfmon counter 2.
F1BH		MSR_C17_PMON_CTR3	Package	Uncore C-box 17 perfmon counter 3.

2.14 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 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28, Table 2-29, Table 2-33, and Table 2-34. For an MSR listed in Table 2-34 that also appears in the model-specific tables of prior generations, Table 2-34 supercede prior generation tables.

Table 2-33 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 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_ STATUS	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0		**Ovf_PMC0**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**

Table 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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**
		34		**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Set 1 to clear Trace_ToPA_PMI. See** Section 35.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved.
		61		**Set 1 to clear Ovf_Uncore**
		62		**Set 1 to clear Ovf_BufDSSAVE**
		63		**Set 1 to clear CondChgd**
560H	1376	IA32_RTIT_OUTPUT_BASE	THREAD	**Trace Output Base Register (R/W)**
		6:0		Reserved.
		MAXPHYADDR[1]-1:7		**Base physical address.**
		63:MAXPHYADDR		Reserved.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	THREAD	**Trace Output Mask Pointers Register (R/W)**
		6:0		Reserved.
		31:7		**MaskOrTableOffset**
		63:32		**Output Offset.**
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		Reserved, MBZ.

Table 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		Reserved, MBZ
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**Reserved; writing 0 will #GP if also setting TraceEn**
		63:14		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**
		0		Reserved, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		63:6		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	THREAD	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
620H		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 2-34 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 2-34. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Enable Package C-State Auto-demotion (R/W)**
		30		**Enable Package C-State Undemotion (R/W)**
		63:31		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.

Table 2-34. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6core active.
		63:48		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28, Table 2-29, Table 2-33 for other MSR definitions applicable to processors with CPUID signature 06_3DH.				

2.15 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 2-35 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 2.15.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-25.
		1		Enable_PPIN (R/W) See Table 2-25.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** See Table 2-25.
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 2-25.
		22:16		Reserved.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23	Package	**PPIN_CAP (R/O)** See Table 2-25.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** See Table 2-25.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** See Table 2-25.
		30	Package	**Programmable TJ OFFSET (R/O)** See Table 2-25.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** See Table 2-25.
		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 MWAT(C1) to MWAIT(C6)
		24:17		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		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 2-25.
		27:24		**TCC Activation Offset (R/W)** See Table 2-25.
		63:28		Reserved.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** Energy consumed by DRAM devices

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
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_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Power Budget Management Status (R0)** When set, frequency is reduced below the operating system request due to PBM limit
		3		**Platform Configuration Services Status (R0)** When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
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.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x00: no monitoring** **0x01: L3 occupancy monitoring** **0x02: Total memory bandwidth monitoring** **0x03: Local memory bandwidth monitoring** **All other encoding reserved**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W)**
		9:0		**RMID**
		31:10		**Reserved**
		51:32		**COS (R/W).**
		63: 52		**Reserved**
C90H	3216	IA32_L3_QOS_MASK_0	Package	**L3 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		**CBM: Bit vector of available L3 ways for COS 0 enforcement**
		63:20		**Reserved**
C91H	3217	IA32_L3_QOS_MASK_1	Package	**L3 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		**CBM: Bit vector of available L3 ways for COS 1 enforcement**
		63:20		**Reserved**
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**

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C95H	3221	IA32_L3_QOS_MASK_5	Package	**L3 Class Of Service Mask - COS 5 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		**CBM: Bit vector of available L3 ways for COS 5 enforcement**
		63:20		**Reserved**
C96H	3222	IA32_L3_QOS_MASK_6	Package	**L3 Class Of Service Mask - COS 6 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		**CBM: Bit vector of available L3 ways for COS 6 enforcement**
		63:20		**Reserved**
C97H	3223	IA32_L3_QOS_MASK_7	Package	**L3 Class Of Service Mask - COS 7 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		**CBM: Bit vector of available L3 ways for COS 7 enforcement**
		63:20		**Reserved**
C98H	3224	IA32_L3_QOS_MASK_8	Package	**L3 Class Of Service Mask - COS 8 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8
		0:19		**CBM: Bit vector of available L3 ways for COS 8 enforcement**
		63:20		**Reserved**
C99H	3225	IA32_L3_QOS_MASK_9	Package	**L3 Class Of Service Mask - COS 9 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		**CBM: Bit vector of available L3 ways for COS 9 enforcement**
		63:20		**Reserved**
C9AH	3226	IA32_L3_QOS_MASK_10	Package	**L3 Class Of Service Mask - COS 10 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		**CBM: Bit vector of available L3 ways for COS 10 enforcement**
		63:20		**Reserved**
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

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0:19		**CBM: Bit vector of available L3 ways for COS 13 enforcement**
		63:20		**Reserved**
C9EH	3230	IA32_L3_QOS_MASK_14	Package	**L3 Class Of Service Mask - COS 14 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		**CBM: Bit vector of available L3 ways for COS 14 enforcement**
		63:20		**Reserved**
C9FH	3231	IA32_L3_QOS_MASK_15	Package	**L3 Class Of Service Mask - COS 15 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		**CBM: Bit vector of available L3 ways for COS 15 enforcement**
		63:20		**Reserved**

2.15.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-36 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 2-19, Table 2-28, Table 2-33, Table 2-35, and Table 2-36.

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		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 2-36. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	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 error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
See Table 2-19, Table 2-28, Table 2-33, and Table 2-35 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.

2.15.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-36 are available to Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID DisplayFamily_DisplayModel = 06_4FH). The Intel® Xeon® processor E5 v4 family is based on the Broadwell

microarchitecture and supports the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-28, Table 2-33, Table 2-35, and Table 2-37.

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
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 2-19, Table 2-20, Table 2-28, and Table 2-29 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.

2.16 MSRS IN THE 6TH GENERATION INTEL® CORE™ PROCESSORS, INTEL® XEON® PROCESSOR SCALABLE FAMILY, 7TH GENERATION INTEL® CORE™ PROCESSORS, AND FUTURE INTEL® CORE™ 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 and have CPUID DisplayFamily_DisplayModel signatures of 06_8EH and 06_9EH. Future Intel® Core™ processors are based on Cannon Lake microarchitecture and have a CPUID DisplayFamily_DisplayModel signature of 06_66H. These processors support the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-24, Table 2-28, Table 2-34, Table 2-38, and Table 2-39. For an MSR listed in Table 2-38 that also appears in the model-specific tables of prior generations, Table 2-38 supercede prior generation tables.

The notation of "Platform" in the Scope column (with respect to MSR_PLATFORM_ENERGY_COUNTER and MSR_PLATFORM_POWER_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor's implementation.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	**MTRR Capality (RO, Architectural).** See Table 2-2
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		18:2		Reserved.
		19		**Disable Race to Halt Optimization (R/W)** Setting this bit disables the Race to Halt optimization and avoid this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization. Default value is 1 for processors that do not support Race to Halt optimization.
		20		**Disable Energy Efficiency Optimization (R/W)** Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.
		63:21		Reserved.
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)**

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		54:35		Reserved.
		55	Thread	**Trace_ToPA_PMI.**
		57:56		Reserved.
		58	Thread	**LBR_Frz.**
		59	Thread	**CTR_Frz.**
		60	Thread	**ASCI.**
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
390H	912	IA32_PERF_GLOBAL_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**

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
391H	913	IA32_PERF_GLOBAL_STATUS_SET		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Set 1 to cause Ovf_PMC0 = 1**
		1	Thread	**Set 1 to cause Ovf_PMC1 = 1**
		2	Thread	**Set 1 to cause Ovf_PMC2 = 1**
		3	Thread	**Set 1 to cause Ovf_PMC3 = 1**
		4	Thread	**Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to cause Ovf_FixedCtr0 = 1**
		33	Thread	**Set 1 to cause Ovf_FixedCtr1 = 1**
		34	Thread	**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55	Thread	**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.
		58	Thread	**Set 1 to cause LBR_Frz = 1**
		59	Thread	**Set 1 to cause CTR_Frz = 1**
		60	Thread	**Set 1 to cause ASCI = 1**
		61	Thread	**Set 1 to cause Ovf_Uncore**
		62	Thread	**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	913	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F7H	1015	MSR_PEBS_FRONTEND	Thread	**FrontEnd Precise Event Condition Select (R/W)**
		2:0		**Event Code Select**
		3		Reserved.
		4		**Event Code Select High**
		7:5		Reserved.
		19:8		**IDQ_Bubble_Length Specifier**
		22:20		**IDQ_Bubble_Width Specifier**
		63:23		**Reserved**
500H	1280	IA32_SGX_SVN_STATUS	Thread	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		15:1		Reserved.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Thread	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Thread	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		48:32		**PacketByteCnt**
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Thread	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Thread	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Thread	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Thread	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Thread	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
64DH	1613	MSR_PLATFORM_ENERGY_COUNTER	Platform*	Platform Energy Counter. (R/O). This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.
		31:0		Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, Included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Enery_Status_Unit.
		63:32		Reserved.
64EH	1614	MSR_PPERF	Thread	Productive Performance Count. (R/O).
		63:0		Hardware's view of workload scalability. See Section 14.4.5.1
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		4		**Residency State Regulation Status (R0)** When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).
		7		**VR Therm Design Current Status (R0)** When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		**Other Status (R0)** When set, frequency is reduced below the operating system request due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		20		**Residency State Regulation Log** When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		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).**

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**PKG_Cx_Monitor.** **Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY**
		63: 3		**Reserved**
653H	1619	MSR_CORE_HDC_ RESIDENCY	Core	Core HDC Idle Residency. (R/O).
		63:0		Core_Cx_Duty_Cycle_Cnt.
655H	1621	MSR_PKG_HDC_SHALLOW_ RESIDENCY	Package	Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle. (R/O).
		63:0		Pkg_C2_Duty_Cycle_Cnt.
656H	1622	MSR_PKG_HDC_DEEP_ RESIDENCY	Package	Package Cx HDC Idle Residency. (R/O).
		63:0		Pkg_Cx_Duty_Cycle_Cnt.
658H	1624	MSR_WEIGHTED_CORE_C0	Package	Core-count Weighted C0 Residency. (R/O).
		63:0		Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.
659H	1625	MSR_ANY_CORE_C0	Package	Any Core C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.
65AH	1626	MSR_ANY_GFXE_C0	Package	Any Graphics Engine C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.
65BH	1627	MSR_CORE_GFXE_OVERLAP_C0	Package	Core and Graphics Engine Overlapped C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.
65CH	1628	MSR_PLATFORM_POWER_LIMIT	Platform*	**Platform Power Limit Control (R/W-L)** Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	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)
		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.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
690H	1680	MSR_ LASTBRANCH_16_FROM_IP	Thread	**Last Branch Record 16 From IP (R/W)** One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.12
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 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Thread	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Thread	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Thread	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Thread	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Thread	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Thread	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Thread	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Thread	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Thread	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Thread	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Thread	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Thread	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W) (frequency refers to processor graphics frequency)**

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.
		12		**Inefficient Operation Status (R0)** When set, processor graphics frequency is operating below target frequency.
		15:13		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Inefficient Operation Log** When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:29		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W)** **(frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.

Table 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		Reserved.
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		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 2-38. 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, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	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: ▪ 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 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Thread	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Thread	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Thread	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Thread	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Thread	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Thread	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Thread	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Thread	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Thread	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Thread	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Thread	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Thread	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP"
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Section 14.4.4, "Managing HWP"
773H	1907	IA32_HWP_INTERRUPT	Thread	See Section 14.4.6, "HWP Notifications"
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP"
		7:0		**Minimum Performance (R/W).**
		15:8		**Maximum Performance (R/W).**
		23:16		**Desired Performance (R/W).**
		31:24		**Energy/Performance Preference (R/W).**
		41:32		**Activity Window (R/W).**
		42		**Package Control (R/W).**
		63:43		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback"
D90H	3472	IA32_BNDCFGS	Thread	See Table 2-2.
DA0H	3488	IA32_XSS	Thread	See Table 2-2.
DB0H	3504	IA32_PKG_HDC_CTL	Package	See Section 14.5.2, "Package level Enabling HDC"
DB1H	3505	IA32_PM_CTL1	Thread	See Section 14.5.

Table 2-39. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors, 7th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
E01H	3585	MSR_UNC_PERF_GLOBAL_ CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
E02H	3586	MSR_UNC_PERF_GLOBAL_ STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.

2.16.1 MSRs Specific to 7th Generation Intel® Core™ Processors based on Kaby Lake Microarchitecture

Table 2-41 lists additional MSRs for 7th generation Intel Core processors with a CPUID DisplayFamily_DisplayModel signature of 06_8EH and 06_9EH. For an MSR listed in Table 2-41 that also appears in the model-specific tables of prior generations, Table 2-41 supersedes prior generation tables.

Table 2-40. Additional MSRs Supported by 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	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 then 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.

Table 2-40. Additional MSRs Supported by 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:8		Reserved.
2F4H	756	MSR_UNCORE_PRMRR_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_PRMRR_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.

2.16.2 MSRs Specific to Future Intel® Core™ Processors

Table 2-41 lists additional MSRs for Future Intel Core processors with a CPUID DisplayFamily_DisplayModel signature of 06_66H. For an MSR listed in Table 2-41 that also appears in the model-specific tables of prior generations, Table 2-41 supersede prior generation tables.

Table 2-41. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		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 result in causing 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.
		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.

Table 2-41. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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', 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.
		25:16		CountWindow The current value of window counter. The count value is frozen on a valid branch monitoring triggering condition. This is an 10-bit unsigned value.
		31:26		Reserved. Reserved for future extension of CountWindow.

Table 2-41. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
		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.

Table 2-41. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
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 count in case that both of the core's thread are in idle state and at least one of the core's thread residency in C1 state or in one of its sub state. The counter is updated only after 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.

2.16.3 MSRs Specific to Intel® Xeon® Processor Scalable Family

Intel® Xeon® Processor Scalable Family (CPUID DisplayFamily_DisplayModel = 06_55H) support the MSRs listed in Table 2-42.

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		18		**SGX global functions enable (R/WL)**

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		20		**LMCE_ON (R/WL)**
		63:21		Reserved.
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-25.
		1		Enable_PPIN (R/W) See Table 2-25.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** See Table 2-25.
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 2-25.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** See Table 2-25.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** See Table 2-25.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** See Table 2-25.
		30	Package	**Programmable TJ OFFSET (R/O)** See Table 2-25.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** See Table 2-25.
		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 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		16		**Automatic C-State Conversion Enable (R/W)** If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6)
		24:17		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** See Table 2-25.
		27:24		**TCC Activation Offset (R/W)** See Table 2-25.
		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: ▪ Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !OC). ▪ 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.
		39:32		RATIO_4 Defines ratio limits.

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
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.

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC0 reports MC error from the IFU module.
401H	1025	IA32_MC0_STATUS	Core	
402H	1026	IA32_MC0_ADDR	Core	
403H	1027	IA32_MC0_MISC	Core	
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC1 reports MC error from the DCU module.
405H	1029	IA32_MC1_STATUS	Core	
406H	1030	IA32_MC1_ADDR	Core	
407H	1031	IA32_MC1_MISC	Core	
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC2 reports MC error from the DTLB module.
409H	1033	IA32_MC2_STATUS	Core	
40AH	1034	IA32_MC2_ADDR	Core	
40BH	1035	IA32_MC2_MISC	Core	
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC3 reports MC error from the MLC module.
40DH	1037	IA32_MC3_STATUS	Core	
40EH	1038	IA32_MC3_ADDR	Core	
40FH	1039	IA32_MC3_MISC	Core	
410H	1040	IA32_MC4_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC4 reports MC error from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from a link interconnect module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the M2M 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the M2M 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC12 report MC error from each channel of a link interconnect module.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** Energy consumed by DRAM devices
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	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_PP0_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**
		31:10		**Reserved**
		51:32		**COS (R/W).**
		63: 52		**Reserved**
C90H	3216	IA32_L3_QOS_MASK_0	Package	**L3 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		**CBM: Bit vector of available L3 ways for COS 0 enforcement**
		63:20		**Reserved**
C91H	3217	IA32_L3_QOS_MASK_1	Package	**L3 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		**CBM: Bit vector of available L3 ways for COS 1 enforcement**
		63:20		**Reserved**

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C92H	3218	IA32_L3_QOS_MASK_2	Package	**L3 Class Of Service Mask - COS 2 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:19		**CBM: Bit vector of available L3 ways for COS 2 enforcement**
		63:20		**Reserved**
C93H	3219	IA32_L3_QOS_MASK_3	Package	**L3 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L3 ways for COS 3 enforcement**
		63:20		**Reserved**
C94H	3220	IA32_L3_QOS_MASK_4	Package	**L3 Class Of Service Mask - COS 4 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4
		0:19		**CBM: Bit vector of available L3 ways for COS 4 enforcement**
		63:20		**Reserved**
C95H	3221	IA32_L3_QOS_MASK_5	Package	**L3 Class Of Service Mask - COS 5 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		**CBM: Bit vector of available L3 ways for COS 5 enforcement**
		63:20		**Reserved**
C96H	3222	IA32_L3_QOS_MASK_6	Package	**L3 Class Of Service Mask - COS 6 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		**CBM: Bit vector of available L3 ways for COS 6 enforcement**
		63:20		**Reserved**
C97H	3223	IA32_L3_QOS_MASK_7	Package	**L3 Class Of Service Mask - COS 7 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		**CBM: Bit vector of available L3 ways for COS 7 enforcement**
		63:20		**Reserved**
C98H	3224	IA32_L3_QOS_MASK_8	Package	**L3 Class Of Service Mask - COS 8 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8
		0:19		**CBM: Bit vector of available L3 ways for COS 8 enforcement**
		63:20		**Reserved**
C99H	3225	IA32_L3_QOS_MASK_9	Package	**L3 Class Of Service Mask - COS 9 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		**CBM: Bit vector of available L3 ways for COS 9 enforcement**
		63:20		**Reserved**
C9AH	3226	IA32_L3_QOS_MASK_10	Package	**L3 Class Of Service Mask - COS 10 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		**CBM: Bit vector of available L3 ways for COS 10 enforcement**
		63:20		**Reserved**

Table 2-42. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C9BH	3227	IA32_L3_QOS_MASK_11	Package	**L3 Class Of Service Mask - COS 11 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11
		0:19		**CBM: Bit vector of available L3 ways for COS 11 enforcement**
		63:20		**Reserved**
C9CH	3228	IA32_L3_QOS_MASK_12	Package	**L3 Class Of Service Mask - COS 12 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12
		0:19		**CBM: Bit vector of available L3 ways for COS 12 enforcement**
		63:20		**Reserved**
C9DH	3229	IA32_L3_QOS_MASK_13	Package	**L3 Class Of Service Mask - COS 13 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13
		0:19		**CBM: Bit vector of available L3 ways for COS 13 enforcement**
		63:20		**Reserved**
C9EH	3230	IA32_L3_QOS_MASK_14	Package	**L3 Class Of Service Mask - COS 14 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		**CBM: Bit vector of available L3 ways for COS 14 enforcement**
		63:20		**Reserved**
C9FH	3231	IA32_L3_QOS_MASK_15	Package	**L3 Class Of Service Mask - COS 15 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		**CBM: Bit vector of available L3 ways for COS 15 enforcement**
		63:20		**Reserved**

2.17 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND FUTURE INTEL® XEON PHI™ PROCESSOR

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily_DisplayModel signature 06_57H, supports the MSR interfaces listed in Table 2-43. These processors are based on the Knights Landing microarchitecture. Future Intel® Xeon Phi™ Processor, with CPUID DisplayFamily_DisplayModel signature 06_85H, supports the MSR interfaces listed in Table 2-43 and Table 2-44. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
6H	6	IA32_MONITOR_FILTER_ SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_ COUNTER	Thread	See Section 17.17, "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)**
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		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'

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	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)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Package	**C-State Configuration Control (R/W)**

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest C-state for the package. This feature does not limit the processor core C-state. The power-on default value from bit[2:0] of this register reports the deepest package C-state the processor is capable to support when manufactured. It is recommended that BIOS always read the power-on default value reported from this bit field to determine the supported deepest C-state on the processor and leave it as default without changing it. 000b - C0/C1 (No package C-state support) 001b - C2 010b - C6 (non retention)* 011b - C6 (Retention)* 100b - Reserved 101b - Reserved 110b - Reserved 111b - No package C-state limit. All C-States supported by the processor are available. Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO registers at MSR_PMG_IO_CAPTURE_BASE[15:0] to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (RO)** When set, locks bits [15:0] of this register for further writes until the next reset occurs.
		25		Reserved.
		26		**C1 State Auto Demotion Enable (R/W)** When set, processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Reserved.
		28		**C1 State Auto Undemotion Enable (R/W)** When set, enables Undemotion from Demoted C1.
		29		**PKG C-State Auto Demotion Enable (R/W)** When set, enables Package C state demotion.
		63:30		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Tile	**Power Management IO Capture Base (R/W)**

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:0		**LVL_2 Base Address (R/W)** Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.
		22:16		**C-State Range (R/W)** The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.
		63:23		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	**Memory Type Range Register (R)** See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
140H	320	MISC_FEATURE_ENABLES	Thread	**MISC_FEATURE_ENABLES**
		0		Reserved.
		1		**User Mode MONITOR and MWAIT (R/W)** If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL > 0 or in virtual-8086 mode. If MWAIT is executed when CPL > 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	See Table 2-2.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	Thread	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		31:0		**Bank Support (SMM-RO)** One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA).
		55:32		Reserved.
		56		**Targeted SMI (SMM-RO)** Set if targeted SMI is supported.
		57		**SMM_CPU_SVRSTR (SMM-RO)** Set if SMM SRAM save/restore feature is supported.
		58		**SMM_CODE_ACCESS_CHK (SMM-RO)** Set if SMM code access check feature is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved.
186H	390	IA32_PERFEVTSEL0	Thread	Performance Monitoring Event Select Register (R/W) See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_INTERRUPT	Module	**Thermal Interrupt Control (R/W)** See Table 2-2.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
19CH	412	IA32_THERM_STATUS	Module	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)**
		1		**Thermal status log (R/WC0)**
		2		**PROTCHOT # or FORCEPR# status (RO)**
		3		**PROTCHOT # or FORCEPR# log (R/WC0)**
		4		**Critical Temperature status (RO)**
		5		**Critical Temperature status log (R/WC0)**
		6		**Thermal threshold #1 status (RO)**
		7		**Thermal threshold #1 log (R/WC0)**
		8		**Thermal threshold #2 status (RO)**
		9		**Thermal threshold #2 log (R/WC0)**
		10		**Power Limitation status (RO)**
		11		**Power Limitation log (R/WC0)**
		15:12		Reserved.
		22:16		**Digital Readout (RO)**
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)**
		31		**Reading Valid (RO)**
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE	Thread	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable**
		2:1		Reserved.
		3		**Automatic Thermal Control Circuit Enable (R/W)**
		6:4		Reserved.
		7		**Performance Monitoring Available (R)**
		10:8		Reserved.
		11		**Branch Trace Storage Unavailable (RO)**
		12		**Processor Event Based Sampling Unavailable (RO)**
		15:13		Reserved.
		16		**Enhanced Intel SpeedStep Technology Enable (R/W)**
		18		**ENABLE MONITOR FSM (R/W)**
		21:19		Reserved.
		22		**Limit CPUID Maxval (R/W)**
		23		**xTPR Message Disable (R/W)**
		33:24		Reserved.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**XD Bit Disable (R/W)**
		37:35		Reserved.
		38		**Turbo Mode Disable (R/W)**
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	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.
		15:8	Package	**Maximum Ratio Limit for Group 0** Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.
		20:16	Package	**Number of Incremental Cores Added to Group 1** Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = “group 0 Max ratio limit” - “group ratio delta for group 1”.
		23:21	Package	**Group Ratio Delta for Group 1** An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.
		28:24	Package	**Number of Incremental Cores Added to Group 2** Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = “group 1 Max ratio limit” - “group ratio delta for group 2”.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:29	Package	**Group Ratio Delta for Group 2** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.
		36:32	Package	**Number of Incremental Cores Added to Group 3** Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = "group 2 Max ratio limit" - "group ratio delta for group 3".
		39:37	Package	**Group Ratio Delta for Group 3** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.
		44:40	Package	**Number of Incremental Cores Added to Group 4** Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = "group 3 Max ratio limit" - "group ratio delta for group 4".
		47:45	Package	**Group Ratio Delta for Group 4** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.
		52:48	Package	**Number of Incremental Cores Added to Group 5** Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = "group 4 Max ratio limit" - "group ratio delta for group 5".
		55:53	Package	**Group Ratio Delta for Group 5** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.
		60:56	Package	**Number of Incremental Cores Added to Group 6** Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = "group 5 Max ratio limit" - "group ratio delta for group 6".
		63:61	Package	**Group Ratio Delta for Group 6** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.
1B0H	432	IA32_ENERGY_PERF_BIAS	Thread	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_ STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_ INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)**
		0		**LBR** Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.
		1		**BTF** Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.
		5:2		Reserved.
		6		**TR** Setting this bit to 1 enables branch trace messages to be sent.
		7		**BTS** Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.
		8		**BTINT** When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.
		9		**BTS_OFF_OS** When set, BTS or BTM is skipped if CPL = 0.
		10		**BTS_OFF_USR** When set, BTS or BTM is skipped if CPL > 0.
		11		**FREEZE_LBRS_ON_PMI** When set, the LBR stack is frozen on a PMI request.
		12		**FREEZE_PERFMON_ON_PMI** When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.
		13		Reserved.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		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.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Core	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Package	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Table 2-2.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O)
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	
		63:0		Package C6 Residency Counter. (R/O)
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	
		63:0		Package C7 Residency Counter. (R/O)
3FCH	1020	MSR_MC0_RESIDENCY	Module	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Module C0 Residency Counter. (R/O)
3FDH	1021	MSR_MC6_RESIDENCY	Module	
		63:0		Module C6 Residency Counter. (R/O)
3FFH	1023	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O)
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter. (R/O)
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
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.
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)** See Table 2-24
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O). See Table 2-24
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O). See Table 2-24
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)** See Table 2-24
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)** See Table 2-24
690H	1680	MSR_CORE_PERF_LIMIT_RE ASONS	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)**

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		7		Reserved.
		8		**Electrical Design Point Status (R0)**
		63:9		Reserved.
6E0H	1760	IA32_TSC_DEADLINE	Core	**TSC Target of Local APIC's TSC Deadline Mode (R/W)** See Table 2-2
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
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)

Table 2-43. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W)** See Table 2-2

Table 2-44 lists model-specific registers that are supported by future Intel® Xeon Phi™ Processors based on the Knights Mill microarchitecture.

Table 2-44. Additional MSRs Supported by Future Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name	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_CTLS	Core	Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.
482H	1154	IA32_VMX_PROCBASED_CTLS	Core	Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)
483H	1155	IA32_VMX_EXIT_CTLS	Core	Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTLS	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_CR0_FIXED0	Core	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2.
487H	1159	IA32_VMX_CR0_FIXED1	Core	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2.
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_CTLS2	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_CTLS	Core	Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O) See Table 2-2.
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Core	Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O) See Table 2-2.
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	Capability Reporting Register of VM-exit Flex Controls (R/O) See Table 2-2.

Table 2-44. Additional MSRs Supported by Future Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	Capability Reporting Register of VM-entry Flex Controls (R/O) See Table 2-2.
491H	1169	IA32_VMX_FMFUNC	Core	Capability Reporting Register of VM-function Controls (R/O) See Table 2-2.

2.18 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-45 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 2-45. MSRs in the Pentium® 4 and Intel® Xeon® Processors

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_LINE_SIZE	3, 4, 6	Shared	See Section 8.10.5, "Monitor/Mwait Address Range Determination."
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."

Table 2-45. 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				
2AH	42	MSR_EBC_HARD_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0			**Output Tri-state Enabled (R)** Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		1			**Execute BIST (R)** Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		2			**In Order Queue Depth (R)** Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		3			**MCERR# Observation Disabled (R)** Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		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.

Table 2-45. 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				
		13:12			**Agent ID (R)** Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		63:14			Reserved.
2BH	43	MSR_EBC_SOFT_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Soft Power-On Configuration (R/W)** Enables and disables processor features.
		0			**RCNT/SCNT On Request Encoding Enable (R/W)** Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).
		1			**Data Error Checking Disable (R/W)** Set to disable system data bus parity checking; clear to enable parity checking.
		2			**Response Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		3			**Address/Request Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		4			**Initiator MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator bus requests (default); clear to enable.
		5			**Internal MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator internal errors (default); clear to enable.
		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.

Table 2-45. 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				
		18:16			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)
					133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
					266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.
		23:19			Reserved.
		31:24			**Core Clock Frequency to System Bus Frequency Ratio (R)** The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	0, 1	Shared	**Processor Frequency Configuration (R)** The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.
		20:0			Reserved.
		23:21			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz All others values reserved.
		63:24			Reserved.

Table 2-45. 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				
3AH	58	IA32_FEATURE_CONTROL	3, 4, 6	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2 (If CPUID.01H:ECX.[bit 5])
79H	121	IA32_BIOS_UPDT_TRIG	0, 1, 2, 3, 4, 6	Shared	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	0, 1, 2, 3, 4, 6	Unique	**BIOS Update Signature ID (R/W)** See Table 2-2.
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	**SMM Monitor Configuration (R/W)** See Table 2-2.
FEH	254	IA32_MTRRCAP	0, 1, 2, 3, 4, 6	Unique	**MTRR Information** See Section 11.11.1, "MTRR Feature Identification.".
174H	372	IA32_SYSENTER_CS	0, 1, 2, 3, 4, 6	Unique	**CS register target for CPL 0 code (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
175H	373	IA32_SYSENTER_ESP	0, 1, 2, 3, 4, 6	Unique	**Stack pointer for CPL 0 stack (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
176H	374	IA32_SYSENTER_EIP	0, 1, 2, 3, 4, 6	Unique	**CPL 0 code entry point (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
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.

Table 2-45. 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				
181H	385	MSR_MCG_RBX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EBX/RBX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
182H	386	MSR_MCG_RCX	0, 1, 2, 3, 4, 6	Unique	**Machine Check ECX/RCX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
183H	387	MSR_MCG_RDX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDX/RDX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
184H	388	MSR_MCG_RSI	0, 1, 2, 3, 4, 6	Unique	**Machine Check ESI/RSI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
185H	389	MSR_MCG_RDI	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDI/RDI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		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."

Table 2-45. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
189H	393	MSR_MCG_RIP	0, 1, 2, 3, 4, 6	Unique	**Machine Check EIP/RIP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
18AH	394	MSR_MCG_MISC	0, 1, 2, 3, 4, 6	Unique	**Machine Check Miscellaneous** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		0			**DS** When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.
		63:1			Reserved.
18BH - 18FH	395	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved.
190H	400	MSR_MCG_R8	0, 1, 2, 3, 4, 6	Unique	**Machine Check R8** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		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."

Table 2-45. 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.
193H	403	MSR_MCG_R11	0, 1, 2, 3, 4, 6	Unique	**Machine Check R11** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
194H	404	MSR_MCG_R12	0, 1, 2, 3, 4, 6	Unique	**Machine Check R12** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
195H	405	MSR_MCG_R13	0, 1, 2, 3, 4, 6	Unique	**Machine Check R13** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
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."

Table 2-45. 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				
199H	409	IA32_PERF_CTL	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
19AH	410	IA32_CLOCK_MODULATION	0, 1, 2, 3, 4, 6	Unique	**Thermal Monitor Control (R/W)** See Table 2-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	**Thermal Interrupt Control (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19CH	412	IA32_THERM_STATUS	0, 1, 2, 3, 4, 6	Shared	**Thermal Monitor Status (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19DH	413	MSR_THERM2_CTL			Thermal Monitor 2 Control.
			3,	Shared	For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.
			4, 6	Shared	For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4, 6	Shared	**Enable Miscellaneous Processor Features (R/W)**
		0			Fast-Strings Enable. See Table 2-2.
		1			Reserved.
		2			**x87 FPU Fopcode Compatibility Mode Enable**
		3			**Thermal Monitor 1 Enable** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
		4			**Split-Lock Disable** When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
					This debug feature is specific to the Pentium 4 processor.
		5			Reserved.

Table 2-45. 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				
		6			**Third-Level Cache Disable (R/W)** When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache. Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs. See Section 11.5.4, "Disabling and Enabling the L3 Cache."
		7			**Performance Monitoring Available (R)** See Table 2-2.
		8			**Suppress Lock Enable** When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			**Prefetch Queue Disable** When set, disables the prefetch queue. When clear (default), enables the prefetch queue.
		10			**FERR# Interrupt Reporting Enable (R/W)** When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.
					When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.
					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.

Table 2-45. 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				
		13	3		**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		17:14			Reserved.
		18	3, 4, 6		**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19			**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector.
					Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			Reserved.
		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.

Table 2-45. 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				
		24			**L1 Data Cache Context Mode (R/W)** When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 11.5.6, "L1 Data Cache Context Mode." When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].
		33:25			Reserved.
		34		Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35			Reserved.
1A1H	417	MSR_PLATFORM_BRV	3, 4, 6	Shared	**Platform Feature Requirements (R)**
		17:0			Reserved.
		18			**PLATFORM Requirements** When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
		63:19			Reserved.
1D7H	471	MSR_LER_FROM_LIP	0, 1, 2, 3, 4, 6	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13.3, "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."

Table 2-45. 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				
		31:0			**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.
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	**Last Branch Record 0 (R/O)** One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH.
					See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
1DCH	477	MSR_LASTBRANCH_1	0, 1, 2	Unique	**Last Branch Record 1** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	**Last Branch Record 2** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	**Last Branch Record 3** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Base MTRR** See Section 11.11.2.3, "Variable Range MTRRs."

Table 2-45. 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				
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs".
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20AH	522	IA32_MTRR_PHYSBASE5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20BH	523	IA32_MTRR_PHYSMASK5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20CH	524	IA32_MTRR_PHYSBASE6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20DH	525	IA32_MTRR_PHYSMASK6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20EH	526	IA32_MTRR_PHYSBASE7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20FH	527	IA32_MTRR_PHYSMASK7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
250H	592	IA32_MTRR_FIX64K_00000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
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."

Table 2-45. 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				
269H	617	IA32_MTRR_FIX4K_C8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26AH	618	IA32_MTRR_FIX4K_D0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26BH	619	IA32_MTRR_FIX4K_D8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26CH	620	IA32_MTRR_FIX4K_E0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26DH	621	IA32_MTRR_FIX4K_E8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26EH	622	IA32_MTRR_FIX4K_F0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26FH	623	IA32_MTRR_FIX4K_F8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
277H	631	IA32_PAT	0, 1, 2, 3, 4, 6	Unique	**Page Attribute Table** See Section 11.11.2.2, "Fixed Range MTRRs."
2FFH	767	IA32_MTRR_DEF_TYPE	0, 1, 2, 3, 4, 6	Shared	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "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."

Table 2-45. 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				
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."
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."

Table 2-45. 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				
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."
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_SAAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AFH	943	MSR_SAAT_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."

Table 2-45. 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				
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.
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."

Table 2-45. 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				
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 19-36.
		23:13			Reserved.
		24			**UOP Tag** Enables replay tagging when set.
		25			**ENABLE_PEBS_MY_THR (R/W)** Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "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 19-36.
400H	1024	IA32_MC0_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-45. 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				
402H	1026	IA32_MC0_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC0_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC		Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-45. 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				
40BH	1035	IA32_MC2_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
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.

Table 2-45. 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				
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
483H	1155	IA32_VMX_EXIT_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls," and see Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls," and see Table 2-2.
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data," and see Table 2-2.
486H	1158	IA32_VMX_CR0_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
487H	1159	IA32_VMX_CR0_FIXED1	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
488H	1160	IA32_VMX_CR4_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
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.

Table 2-45. 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				
48BH	1163	IA32_VMX_PROCBASED_CTLS2	3, 4, 6	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
600H	1536	IA32_DS_AREA	0, 1, 2, 3, 4, 6	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "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.
					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.

Table 2-45. 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				
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	3, 4, 6	Unique	**Last Branch Record 0 (R/W)** One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 17.12, "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.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	**Last Branch Record 9** See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	**Last Branch Record 10** See description of MSR_LASTBRANCH_0 at 6C0H.

Table 2-45. 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				
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 6C0H.
C000_0080H		IA32_EFER	3, 4, 6	Unique	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	3, 4, 6	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	3, 4, 6	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	3, 4, 6	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	3, 4, 6	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	3, 4, 6	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	3, 4, 6	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

NOTES

1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.

2.18.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-46 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 2-46. 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)**
107D0H		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 2-47 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 2-47 are shared between logical processors in the same core, but are replicated for each core.

Table 2-47. 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."
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)**

2.19 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-48. 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 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
1H	1	P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and see Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and see Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		6: 5		Reserved
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		11		Reserved
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		13		Reserved
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		**APIC Cluster ID (R/O)**

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		18		**System Bus Frequency (R/O)** 0 = 100 MHz 1 = Reserved
		19		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)**
		26:22		**Clock Frequency Ratio (R/O)**
3AH	58	IA32_FEATURE_CONTROL	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2.
40H	64	MSR_LASTBRANCH_0	Unique	**Last Branch Record 0 (R/W)** One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the 'to' address. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.15, "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:

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count. (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count. (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	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.

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		**EIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_ MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_ INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	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 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		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 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
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 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MC0_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	MSR_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	MSR_MC3_MISC	Unique	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).

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
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 ADDRV 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 9])
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9])
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9])
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls" (If CPUID.01H:ECX.[bit 9])
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls" (If CPUID.01H:ECX.[bit 9])
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data" (If CPUID.01H:ECX.[bit 9])
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0" (If CPUID.01H:ECX.[bit 9])
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0" (If CPUID.01H:ECX.[bit 9])
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4" (If CPUID.01H:ECX.[bit 9])

Table 2-48. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4" (If CPUID.01H:ECX.[bit 9])
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration" (If CPUID.01H:ECX.[bit 9])
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9] and IA32_VMX_PROCBASED_CTLS[bit 63])
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "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.

2.20 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.21 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

Table 2-49. MSRs in Pentium M Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.17, "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.

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
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.
		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.

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		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: ▪ 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.
		0	**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		4:1	Reserved.

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		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.
		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.

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		63:3	Reserved.
198H	408	IA32_PERF_STATUS	See Table 2-2.
199H	409	IA32_PERF_CTL	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	**Clock Modulation (R/W).** See Table 2-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19DH	413	MSR_THERM2_CTL	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	**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.

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		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.
1C9H	457	MSR_LASTBRANCH_TOS	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: ▪ MSR_LASTBRANCH_0_FROM_IP (at 40H) ▪ Section 17.15, "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)."

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
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_MC0_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	See Section 14.3.2.3., "IA32_MCi_ADDR MSRs". The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	See Chapter 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-49. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
40EH	1038	MSR_MC4_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	MSR_MC3_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	MSR_MC3_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
600H	1536	IA32_DS_AREA	**DS Save Area (R/W)** See Table 2-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.6.3.4, "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.

2.21 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

Table 2-50. MSRs in the P6 Family Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."
17H	23	IA32_PLATFORM_ID	**Platform ID (R)** The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
		49:0	Reserved.

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		52:50	**Platform Id (R)** Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
		56:53	L2 Cache Latency Read.
		59:57	Reserved.
		60	Clock Frequency Ratio Read.
		63:61	Reserved.
1BH	27	APIC_BASE	Section 10.4.4, "Local APIC Status and Location."
		7:0	Reserved.
		8	**Boot Strap Processor indicator Bit** 1 = BSP
		10:9	Reserved.
		11	**APIC Global Enable Bit - Permanent till reset** 1 = Enabled 0 = Disabled
		31:12	APIC Base Address.
		63:32	Reserved.
2AH	42	EBL_CR_POWERON	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0	Reserved.[1]
		1	**Data Error Checking Enable (R/W)** 1 = Enabled 0 = Disabled
		2	**Response Error Checking Enable FRCERR Observation Enable (R/W)** 1 = Enabled 0 = Disabled
		3	**AERR# Drive Enable (R/W)** 1 = Enabled 0 = Disabled
		4	**BERR# Enable for Initiator Bus Requests (R/W)** 1 = Enabled 0 = Disabled
		5	Reserved.

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		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)**
		27	**Clock Frequency Ratio**
		63:28	Reserved.[1]
33H	51	TEST_CTL	**Test Control Register**

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		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] BBL_CR_ADDR [63:32] BBL_CR_ADDR [31:3] BBL_CR_ADDR [2:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses. Reserved, Address bits [35:3] Reserved Set to 0.
118H	280	BBL_CR_DECC[63:0]	Data ECC register D[7:0]: used to write ECC and read ECC to/from L2
119H	281	BBL_CR_CTL BL_CR_CTL[63:22] BBL_CR_CTL[21]	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response Reserved Processor number[2] Disable = 1 Enable = 0 Reserved
		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]	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

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL[4:0] 01100 01110 01111 00010 00011 010 + MESI encode 111 + MESI encode 100 + MESI encode	L2 Command Data Read w/ LRU update (RLU) Tag Read w/ Data Read (TRR) Tag Inquire (TI) L2 Control Register Read (CR) L2 Control Register Write (CW) Tag Write w/ Data Read (TWR) Tag Write w/ Data Write (TWW) Tag Write (TW)
11AH	282	BBL_CR_TRIG	Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.
11BH	283	BBL_CR_BUSY	Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY
11EH	286	BBL_CR_CTL3 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 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13 00001 00010 00100 01000 10000 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 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		18	**E** Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	**PC** Enabled the signaling of performance counter overflow via BP0 pin
		20	**INT** Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		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 BP0 pin.

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	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	DEBUGCTLMSR	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 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	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	MC0_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	MC0_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 MC0_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
402H	1026	MC0_ADDR	
403H	1027	MC0_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

Table 2-50. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MC0_STATUS.
406H	1030	MC1_ADDR	
407H	1031	MC1_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
408H	1032	MC2_CTL	
409H	1033	MC2_STATUS	Bit definitions same as MC0_STATUS.
40AH	1034	MC2_ADDR	
40BH	1035	MC2_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
40CH	1036	MC4_CTL	
40DH	1037	MC4_STATUS	Bit definitions same as MC0_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.
40EH	1038	MC4_ADDR	Defined in MCA architecture but not implemented in P6 Family processors.
40FH	1039	MC4_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
410H	1040	MC3_CTL	
411H	1041	MC3_STATUS	Bit definitions same as MC0_STATUS.
412H	1042	MC3_ADDR	
413H	1043	MC3_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

NOTES

1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL_CR_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL_CR_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

2.22 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5_MC_ADDR, P5_MC_TYPE, and TSC MSRs (named IA32_P5_MC_ADDR, IA32_P5_MC_TYPE, and IA32_TIME_STAMP_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

Table 2-51. 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	CTR0	Section 18.6.9.3, "Events Counted."
13H	19	CTR1	Section 18.6.9.3, "Events Counted."

2.23 MSR INDEX

MSRs of recent processors are indexed here for convenience. IA32 MSRs are excluded from this index.

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_B0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_B0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_B0_PMON_MASK	
06_2EH	See Table 2-16
MSR_B0_PMON_MATCH	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR0	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR1	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR2	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR3	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_B1_PMON_MASK	
06_2EH	See Table 2-16
MSR_B1_PMON_MATCH	
06_2EH	See Table 2-16
MSR_BBL_CR_CTL	
06_09H	See Table 2-49
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 2-48
06_09H	See Table 2-49

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_BPU_CCCR0	
0FH	See Table 2-45
MSR_BPU_CCCR1	
0FH	See Table 2-45
MSR_BPU_CCCR2	
0FH	See Table 2-45
MSR_BPU_CCCR3	
0FH	See Table 2-45
MSR_BPU_COUNTER0	
0FH	See Table 2-45
MSR_BPU_COUNTER1	
0FH	See Table 2-45
MSR_BPU_COUNTER2	
0FH	See Table 2-45
MSR_BPU_COUNTER3	
0FH	See Table 2-45
MSR_BPU_ESCR0	
0FH	See Table 2-45
MSR_BPU_ESCR1	
0FH	See Table 2-45
MSR_BR_DETECT_COUNTER_CONFIG_i	
06_66H	See Table 2-41
MSR_BR_DETECT_CTRL	
06_66H	See Table 2-41
MSR_BR_DETECT_STATUS	
06_66H	See Table 2-41
MSR_BSU_ESCR0	
0FH	See Table 2-45
MSR_BSU_ESCR1	
0FH	See Table 2-45
MSR_C0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C0_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_OVF_CTRL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
MSR_C0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C0_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR2	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C1_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C1_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C1_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C1_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C1_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C10_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C10_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C10_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C11_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C11_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C11_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C12_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C12_PMON_BOX_FILTER0	
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C12_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C13_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C13_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C13_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C14_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C14_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C14_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR1	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_CTL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR1	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C2_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C2_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C2_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C2_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C2_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C2_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C3_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C3_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C3_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C3_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C3_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C3_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C3_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C3_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C4_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C4_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_FILTER1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C4_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C4_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C4_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C4_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C4_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C5_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C5_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C5_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C5_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C5_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C5_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C5_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C6_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C6_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C6_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C6_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C6_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C6_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C6_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C6_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C7_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C7_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_FILTER1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C7_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C7_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C7_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C7_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C7_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C8_PMON_BOX_CTRL	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C8_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-18
MSR_C8_PMON_BOX_STATUS	
06_2FH	See Table 2-18
06_3FH	See Table 2-32
MSR_C8_PMON_CTR0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR4	
06_2FH	See Table 2-18
MSR_C8_PMON_CTR5	
06_2FH	See Table 2-18
MSR_C8_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL4	
06_2FH	See Table 2-18
MSR_C8_PMON_EVNT_SEL5	
06_2FH	See Table 2-18
MSR_C9_PMON_BOX_CTRL	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C9_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-18
MSR_C9_PMON_BOX_STATUS	
06_2FH	See Table 2-18
06_3FH	See Table 2-32
MSR_C9_PMON_CTR0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C9_PMON_CTR2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR4	
06_2FH	See Table 2-18
MSR_C9_PMON_CTR5	
06_2FH	See Table 2-18
MSR_C9_PMON_EVNT_SEL0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL4	
06_2FH	See Table 2-18
MSR_C9_PMON_EVNT_SEL5	
06_2FH	See Table 2-18
MSR_CC6_DEMOTION_POLICY_CONFIG	
06_37H	See Table 2-9
MSR_CONFIG_TDP_CONTROL	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-43
MSR_CONFIG_TDP_LEVEL1	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_57H	See Table 2-43
MSR_CONFIG_TDP_LEVEL2	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-43
MSR_CONFIG_TDP_NOMINAL	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-43
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 2-41
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 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
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 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-43
MSR_CORE_C7_RESIDENCY	
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
MSR_CORE_GFXE_OVERLAP_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_CORE_HDC_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_CORE_PERF_LIMIT_REASONS	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
MSR_CORE_THREAD_COUNT	
06_3FH	See Table 2-31
MSR_CRU_ESCR0	
0FH	See Table 2-45
MSR_CRU_ESCR1	
0FH	See Table 2-45
MSR_CRU_ESCR2	
0FH	See Table 2-45
MSR_CRU_ESCR3	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
MSR_EBC_SOFT_POWERON	
0FH	See Table 2-45
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 2-48
06_09H	See Table 2-49
MSR_EFSB_DRDY0	
0F_03H, 0F_04H	See Table 2-46
MSR_EFSB_DRDY1	
0F_03H, 0F_04H	See Table 2-46
MSR_EMON_L3_CTR_CTL0	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL1	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL2	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL3	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL4	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL5	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL6	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_CTR_CTL7	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-47
MSR_EMON_L3_GL_CTL	
06_0FH, 06_17H	See Table 2-3
MSR_ERROR_CONTROL	
06_2DH	See Table 2-22
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3F	See Table 2-31
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 2-17
06_2FH	See Table 2-18
06_2AH, 06_2DH.	See Table 2-19
06_57H	See Table 2-43
MSR_FIRM_ESCR0	
0FH	See Table 2-45
MSR_FIRM_ESCR1	
0FH	See Table 2-45
MSR_FLAME_CCCR0	
0FH	See Table 2-45
MSR_FLAME_CCCR1	
0FH	See Table 2-45
MSR_FLAME_CCCR2	
0FH	See Table 2-45
MSR_FLAME_CCCR3	
0FH	See Table 2-45
MSR_FLAME_COUNTER0	
0FH	See Table 2-45
MSR_FLAME_COUNTER1	
0FH	See Table 2-45
MSR_FLAME_COUNTER2	
0FH	See Table 2-45
MSR_FLAME_COUNTER3	
0FH	See Table 2-45
MSR_FLAME_ESCR0	
0FH	See Table 2-45
MSR_FLAME_ESCR1	
0FH	See Table 2-45
MSR_FSB_ESCR0	
0FH	See Table 2-45
MSR_FSB_ESCR1	
0FH	See Table 2-45
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 2-48
MSR_GQ_SNOOP_MESF	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
MSR_IS_ESCR0	
0FH	See Table 2-45
MSR_IS_ESCR1	
0FH	See Table 2-45
MSR_ITLB_ESCR0	
0FH	See Table 2-45
MSR_ITLB_ESCR1	
0FH	See Table 2-45
MSR_IX_ESCR0	
0FH	See Table 2-45
MSR_IX_ESCR1	
0FH	See Table 2-45
MSR_LASTBRANCH_0	
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_10_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_10_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_11_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_11_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_12_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_12_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_13_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_13_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_14_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_14_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_15_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_15_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH.	See Table 2-45
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.	See Table 2-38
MSR_LASTBRANCH_16_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_17_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_17_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_18_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_18_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_19_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_19_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_2	
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
MSR_LASTBRANCH_2_FROM_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_2_TO_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_20_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
MSR_LASTBRANCH_22_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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
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	See Table 2-38
MSR_LASTBRANCH_29_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	See Table 2-38
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	See Table 2-38
MSR_LASTBRANCH_3	
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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	See Table 2-38
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	See Table 2-38
MSR_LASTBRANCH_31_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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	See Table 2-38
MSR_LASTBRANCH_4	
06_0EH	See Table 2-48
06_09H	See Table 2-49
MSR_LASTBRANCH_4_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_5	
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_6	
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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

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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_7	
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_8_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_8_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_9_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_9_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-45
MSR_LASTBRANCH_TOS	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_57H	See Table 2-43
06_0EH	See Table 2-48
06_09H	See Table 2-49
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	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_10	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_11	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_12	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_13	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_14	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_15	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_16	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_17	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_18	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LBR_INFO_19	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_2	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_20	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_21	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_22	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_23	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_24	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_25	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_26	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_27	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_28	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_29	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_3	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_30	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LBR_INFO_31	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_4	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_5	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_6	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_7	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_8	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_9	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-43
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
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-43
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
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

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-43
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
MSR_M0_PMON_ADDR_MASK	
06_2EH	See Table 2-16
MSR_M0_PMON_ADDR_MATCH	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR0	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR1	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR2	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR3	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_M0_PMON_DSP	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M0_PMON_ISS	
06_2EH	See Table 2-16
MSR_M0_PMON_MAP	
06_2EH	See Table 2-16
MSR_M0_PMON_MM_CONFIG	
06_2EH	See Table 2-16
MSR_M0_PMON_MSC_THR	
06_2EH	See Table 2-16
MSR_M0_PMON_PGT	
06_2EH	See Table 2-16
MSR_M0_PMON_PLD	
06_2EH	See Table 2-16
MSR_M0_PMON_TIMESTAMP	
06_2EH	See Table 2-16
MSR_M0_PMON_ZDP	
06_2EH	See Table 2-16
MSR_M1_PMON_ADDR_MASK	
06_2EH	See Table 2-16
MSR_M1_PMON_ADDR_MATCH	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR0	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR1	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR2	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR3	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR4	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR5	
06_2EH	See Table 2-16
MSR_M1_PMON_DSP	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_M1_PMON_ISS	
06_2EH	See Table 2-16
MSR_M1_PMON_MAP	
06_2EH	See Table 2-16
MSR_M1_PMON_MM_CONFIG	
06_2EH	See Table 2-16
MSR_M1_PMON_MSC_THR	
06_2EH	See Table 2-16
MSR_M1_PMON_PGT	
06_2EH	See Table 2-16
MSR_M1_PMON_PLD	
06_2EH	See Table 2-16
MSR_M1_PMON_TIMESTAMP	
06_2EH	See Table 2-16
MSR_M1_PMON_ZDP	
06_2EH	See Table 2-16
IA32_MC0_MISC / MSR_MC0_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_MC0_RESIDENCY	
06_57H	See Table 2-43
IA32_MC1_MISC / MSR_MC1_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
IA32_MC10_ADDR / MSR_MC10_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_CTL / MSR_MC10_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_MISC / MSR_MC10_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_STATUS / MSR_MC10_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC11_ADDR / MSR_MC11_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_CTL / MSR_MC11_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_MISC / MSR_MC11_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_STATUS / MSR_MC11_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC12_ADDR / MSR_MC12_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_CTL / MSR_MC12_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_MISC / MSR_MC12_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_STATUS / MSR_MC12_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_ADDR / MSR_MC13_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_CTL / MSR_MC13_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_MISC / MSR_MC13_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC13_STATUS / MSR_MC13_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_ADDR / MSR_MC14_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_CTL / MSR_MC14_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_MISC / MSR_MC14_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_STATUS / MSR_MC14_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_ADDR / MSR_MC15_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_CTL / MSR_MC15_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC15_MISC / MSR_MC15_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_STATUS / MSR_MC15_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_ADDR / MSR_MC16_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_CTL / MSR_MC16_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_MISC / MSR_MC16_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_STATUS / MSR_MC16_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC17_ADDR / MSR_MC17_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC17_CTL / MSR_MC17_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC17_MISC / MSR_MC17_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC17_STATUS / MSR_MC17_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_ADDR / MSR_MC18_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_CTL / MSR_MC18_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_MISC / MSR_MC18_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC18_STATUS / MSR_MC18_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_ADDR / MSR_MC19_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_CTL / MSR_MC19_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_MISC / MSR_MC19_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_STATUS / MSR_MC19_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC2_MISC / MSR_MC2_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
IA32_MC20_ADDR / MSR_MC20_ADDR	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC20_CTL / MSR_MC20_CTL	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC20_MISC / MSR_MC20_MISC	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC20_STATUS / MSR_MC20_STATUS	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC21_ADDR / MSR_MC21_ADDR	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_CTL / MSR_MC21_CTL	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_MISC / MSR_MC21_MISC	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_STATUS / MSR_MC21_STATUS	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC22_ADDR / MSR_MC22_ADDR	
06_3EH	See Table 2-25
IA32_MC22_CTL / MSR_MC22_CTL	
06_3EH	See Table 2-25
IA32_MC22_MISC / MSR_MC22_MISC	
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC22_STATUS / MSR_MC22_STATUS	
06_3EH	See Table 2-25
IA32_MC23_ADDR / MSR_MC23_ADDR	
06_3EH	See Table 2-25
IA32_MC23_CTL / MSR_MC23_CTL	
06_3EH	See Table 2-25
IA32_MC23_MISC / MSR_MC23_MISC	
06_3EH	See Table 2-25
IA32_MC23_STATUS / MSR_MC23_STATUS	
06_3EH	See Table 2-25
IA32_MC24_ADDR / MSR_MC24_ADDR	
06_3EH	See Table 2-25
IA32_MC24_CTL / MSR_MC24_CTL	
06_3EH	See Table 2-25
IA32_MC24_MISC / MSR_MC24_MISC	
06_3EH	See Table 2-25
IA32_MC24_STATUS / MSR_MC24_STATUS	
06_3EH	See Table 2-25
IA32_MC25_ADDR / MSR_MC25_ADDR	
06_3EH	See Table 2-25
IA32_MC25_CTL / MSR_MC25_CTL	
06_3EH	See Table 2-25
IA32_MC25_MISC / MSR_MC25_MISC	
06_3EH	See Table 2-25
IA32_MC25_STATUS / MSR_MC25_STATUS	
06_3EH	See Table 2-25
IA32_MC26_ADDR / MSR_MC26_ADDR	
06_3EH	See Table 2-25
IA32_MC26_CTL / MSR_MC26_CTL	
06_3EH	See Table 2-25
IA32_MC26_MISC / MSR_MC26_MISC	
06_3EH	See Table 2-25
IA32_MC26_STATUS / MSR_MC26_STATUS	
06_3EH	See Table 2-25
IA32_MC27_ADDR / MSR_MC27_ADDR	
06_3EH	See Table 2-25
IA32_MC27_CTL / MSR_MC27_CTL	
06_3EH	See Table 2-25
IA32_MC27_MISC / MSR_MC27_MISC	
06_3EH	See Table 2-25
IA32_MC27_STATUS / MSR_MC27_STATUS	
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC28_ADDR / MSR_MC28_ADDR	
06_3EH	See Table 2-25
IA32_MC28_CTL / MSR_MC28_CTL	
06_3EH	See Table 2-25
IA32_MC28_MISC / MSR_MC28_MISC	
06_3EH	See Table 2-25
IA32_MC28_STATUS / MSR_MC28_STATUS	
06_3EH	See Table 2-25
IA32_MC29_ADDR / MSR_MC29_ADDR	
06_3EH	See Table 2-26
IA32_MC29_CTL / MSR_MC29_CTL	
06_3EH	See Table 2-26
IA32_MC29_MISC / MSR_MC29_MISC	
06_3EH	See Table 2-26
IA32_MC29_STATUS / MSR_MC29_STATUS	
06_3EH	See Table 2-26
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 2-14
06_57H	See Table 2-43
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_57H	See Table 2-43
06_0EH	See Table 2-48
06_09H	See Table 2-49
IA32_MC3_MISC / MSR_MC3_MISC	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_0EH	See Table 2-48
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 2-14
06_57H	See Table 2-43

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH	See Table 2-48
06_09H	See Table 2-49
IA32_MC30_ADDR / MSR_MC30_ADDR	
06_3EH	See Table 2-26
IA32_MC30_CTL / MSR_MC30_CTL	
06_3EH	See Table 2-26
IA32_MC30_MISC / MSR_MC30_MISC	
06_3EH	See Table 2-26
IA32_MC30_STATUS / MSR_MC30_STATUS	
06_3EH	See Table 2-26
IA32_MC31_ADDR / MSR_MC31_ADDR	
06_3EH	See Table 2-26
IA32_MC31_CTL / MSR_MC31_CTL	
06_3EH	See Table 2-26
IA32_MC31_MISC / MSR_MC31_MISC	
06_3EH	See Table 2-26
IA32_MC31_STATUS / MSR_MC31_STATUS	
06_3EH	See Table 2-26
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 2-14
06_57H	See Table 2-43
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_57H	See Table 2-43
06_0EH	See Table 2-48
06_09H	See Table 2-49
IA32_MC4_CTL2 / MSR_MC4_CTL2	
06_2AH, 06_2DH	See Table 2-19
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 2-14
06_57H	See Table 2-43

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH	See Table 2-48
06_09H	See Table 2-49
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 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-43
06_0EH	See Table 2-48
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 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-43
06_0EH	See Table 2-48
IA32_MC5_MISC / MSR_MC5_MISC	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_0EH	See Table 2-48
IA32_MC5_STATUS / MSR_MC5_STATUS	
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 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-43
06_0EH	See Table 2-48
IA32_MC6_ADDR / MSR_MC6_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC6_CTL / MSR_MC6_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
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 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
MSR_MC6_RESIDENCY_COUNTER	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_37H	See Table 2-9
06_57H	See Table 2-43
IA32_MC6_STATUS / MSR_MC6_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_ADDR / MSR_MC7_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_CTL / MSR_MC7_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_MISC / MSR_MC7_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_STATUS / MSR_MC7_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC8_ADDR / MSR_MC8_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_CTL / MSR_MC8_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_MISC / MSR_MC8_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_STATUS / MSR_MC8_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC9_ADDR / MSR_MC9_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_CTL / MSR_MC9_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_MISC / MSR_MC9_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_STATUS / MSR_MC9_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
MSR_MCG_MISC	
0FH	See Table 2-45
MSR_MCG_R10	
0FH	See Table 2-45
MSR_MCG_R11	
0FH	See Table 2-45
MSR_MCG_R12	
0FH	See Table 2-45
MSR_MCG_R13	
0FH	See Table 2-45
MSR_MCG_R14	
0FH	See Table 2-45
MSR_MCG_R15	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
MSR_MCG_R8	
0FH	See Table 2-45
MSR_MCG_R9	
0FH	See Table 2-45
MSR_MCG_RAX	
0FH	See Table 2-45
MSR_MCG_RBP	
0FH	See Table 2-45
MSR_MCG_RBX	
0FH	See Table 2-45
MSR_MCG_RCX	
0FH	See Table 2-45
MSR_MCG_RDI	
0FH	See Table 2-45
MSR_MCG_RDX	
0FH	See Table 2-45
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5	
0FH	See Table 2-45
MSR_MCG_RFLAGS	
0FH	See Table 2-45
MSR_MCG_RIP	
0FH	See Table 2-45
MSR_MCG_RSI	
0FH	See Table 2-45
MSR_MCG_RSP	
0FH	See Table 2-45
MSR_MISC_FEATURE_CONTROL	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_MISC_PWR_MGMT	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_MOB_ESCR0	
0FH	See Table 2-45
MSR_MOB_ESCR1	
0FH	See Table 2-45
MSR_MS_CCCR0	
0FH	See Table 2-45
MSR_MS_CCCR1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
MSR_MS_CCCR2	
0FH	See Table 2-45
MSR_MS_CCCR3	
0FH	See Table 2-45
MSR_MS_COUNTER0	
0FH	See Table 2-45
MSR_MS_COUNTER1	
0FH	See Table 2-45
MSR_MS_COUNTER2	
0FH	See Table 2-45
MSR_MS_COUNTER3	
0FH	See Table 2-45
MSR_MS_ESCR0	
0FH	See Table 2-45
MSR_MS_ESCR1	
0FH	See Table 2-45
MSR_MTRRCAP	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-43
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 2-17
06_2FH	See Table 2-18
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-43
MSR_PCIE_PLL_RATIO	
06_3FH	See Table 2-31
MSR_PCU_PMON_BOX_CTL	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_BOX_FILTER	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_BOX_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR2	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR3	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL2	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL3	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PEBS_ENABLE	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH	See Table 2-12
06_7AH	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3EH	See Table 2-26
06_57H	See Table 2-43
0FH	See Table 2-45
MSR_PEBS_FRONTEND	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PEBS_LD_LAT	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_PEBS_MATRIX_VERT	
0FH	See Table 2-45
MSR_PEBS_NUM_ALT	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
MSR_PERF_CAPABILITIES	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR_CTRL	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR0	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR1	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR2	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_GLOBAL_CTRL	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_GLOBAL_OVF_CTRL	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_PERF_GLOBAL_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
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 2-19
MSR_PKG_C10_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_45H	See Table 2-29 and Table 2-30
06_4FH	See Table 2-37
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 2-19
06_57H	See Table 2-43
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 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_66H	See Table 2-41
06_57H	See Table 2-43
MSR_PKG_C4_RESIDENCY	
06_27H	See Table 2-5
MSR_PKG_C6_RESIDENCY	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
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 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-43
MSR_PKG_C7_RESIDENCY	
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-43
MSR_PKG_C8_RESIDENCY	
06_45H	See Table 2-30
06_4FH	See Table 2-37
MSR_PKG_C9_RESIDENCY	
06_45H	See Table 2-30
06_4FH	See Table 2-37
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_45H	See Table 2-30
06_3F	See Table 2-31
06_3DH	See Table 2-34
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
MSR_PKG_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 2-19
MSR_PKG_HDC_CONFIG	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_HDC_DEEP_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_HDC_SHALLOW_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_PERF_STATUS	
06_5CH, 06_7AH	See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H	See Table 2-43
MSR_PKG_POWER_INFO	
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 2-19
06_57H	See Table 2-43
MSR_PKG_POWER_LIMIT	
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 2-19
06_57H	See Table 2-43
MSR_PKGC_IRTL1	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H.	See Table 2-28
MSR_PKGC_IRTL2	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H.	See Table 2-28
MSR_PKGC3_IRTL	
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH	See Table 2-19
MSR_PKGC6_IRTL	
06_2AH, 06_2DH	See Table 2-19
MSR_PKGC7_IRTL	
06_2AH	See Table 2-20
MSR_PLATFORM_BRV	
0FH	See Table 2-45
MSR_PLATFORM_ENERGY_COUNTER	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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 2-14
MSR_PLATFORM_INFO	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-28 and Table 2-29
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
MSR_PLATFORM_POWER_LIMIT	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_57H	See Table 2-43
MSR_PMH_ESCR0	
0FH	See Table 2-45
MSR_PMH_ESCR1	
0FH	See Table 2-45
MSR_PMON_GLOBAL_CONFIG	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PMON_GLOBAL_CTL	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PMON_GLOBAL_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_POWER_CTL	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_PP0_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 2-19
06_57H	See Table 2-43
MSR_PP0_POLICY	
06_2AH, 06_45H	See Table 2-20
MSR_PP0_POWER_LIMIT	
06_4CH	See Table 2-11

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-43
MSR_PP1_ENERGY_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PP1_POLICY	
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PP1_POWER_LIMIT	
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PPERF	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PPIN	
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
MSR_PPIN_CTL	
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
MSR_PRMRR_PHYS_BASE	
06_8EH, 06_9EH	See Table 2-40
MSR_PRMRR_PHYS_MASK	
06_8EH, 06_9EH	See Table 2-40
MSR_PRMRR_VALID_CONFIG	
06_8EH, 06_9EH	See Table 2-40
MSR_RING_RATIO_LIMIT	
06_8EH, 06_9EH	See Table 2-40
MSR_R0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_R0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_R0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR0	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR1	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR2	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR3	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR6	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR7	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL6	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL7	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P0	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P1	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P2	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P3	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P4	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P5	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P6	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P7	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P0	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R0_PMON_QLX_P1	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P2	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P3	
06_2EH	See Table 2-16
MSR_R1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_R1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_R1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR10	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR11	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR12	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR13	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR14	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR15	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR8	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR9	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL10	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL11	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL12	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL13	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL14	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL15	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL8	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R1_PMON_EVNT_SEL9	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P10	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P11	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P12	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P13	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P14	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P15	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P8	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P9	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P4	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P5	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P6	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P7	
06_2EH	See Table 2-16
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 2-19
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
MSR_RAT_ESCR0	
0FH	See Table 2-45
MSR_RAT_ESCR1	
0FH	See Table 2-45
MSR_RING_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_S0_PMON_BOX_CTRL	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_S0_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_S0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_S0_PMON_CTR0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_MASK	
06_2EH	See Table 2-16
MSR_S0_PMON_MATCH	
06_2EH	See Table 2-16
MSR_S1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_S1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_S1_PMON_CTR0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_CTR1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_CTR2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_CTR3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_MASK	
06_2EH	See Table 2-16
MSR_S1_PMON_MATCH	
06_2EH	See Table 2-16
MSR_S2_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_S2_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR0	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR1	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR2	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR3	
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_S2_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_S3_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_S3_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR0	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR1	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR2	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR3	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_SAAT_ESCR0	
0FH	See Table 2-45
MSR_SAAT_ESCR1	
0FH	See Table 2-45
MSR_SGXOWNEREPOCH0	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_SGXOWNEREPOCH1	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
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 2-14
06_2AH, 06_2DH	See Table 2-19

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_57H	See Table 2-43
MSR_SMM_BLOCKED	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_DELAYED	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_FEATURE_CONTROL	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_MCA_CAP	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
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 2-45
MSR_TBPU_ESCR0	
0FH	See Table 2-45
MSR_TBPU_ESCR1	
0FH	See Table 2-45
MSR_TC_ESCR0	
0FH	See Table 2-45
MSR_TC_ESCR1	
0FH	See Table 2-45
MSR_TC_PRECISE_EVENT	
0FH	See Table 2-45
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 2-14
06_2AH, 06_2DH	See Table 2-19
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-43
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

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-45
06_0EH	See Table 2-48
06_09H	See Table 2-49
MSR_THREAD_ID_INFO	
06_3FH	See Table 2-31
MSR_TRACE_HUB_STH_ACPIBAR_BASE	
06_8EH, 06_9EH	See Table 2-40
MSR_TURBO_ACTIVATION_RATIO	
06_5CH, 06_7AH	See Table 2-12
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-43
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 2-14
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 2-14
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
06_2EH	See Table 2-16
06_25H, 06_2CH	See Table 2-17
06_2FH	See Table 2-18
06_2AH, 06_45H	See Table 2-20
06_2DH	See Table 2-22
06_3EH	See Table 2-25 and Table 2-26
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3FH	See Table 2-31
06_3DH	See Table 2-34
06_56H, 06_4FH	See Table 2-35
06_55H	See Table 2-42
06_57H	See Table 2-43
MSR_TURBO_RATIO_LIMIT1	
06_3EH	See Table 2-25 and Table 2-26
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
MSR_TURBO_RATIO_LIMIT2	
06_3FH	See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_TURBO_RATIO_LIMIT3	
06_56H	See Table 2-36
06_4FH	See Table 2-37
MSR_TURBO_RATIO_LIMIT_CORES	
06_55H	See Table 2-42
MSR_U_PMON_BOX_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_U_PMON_CTR	
06_2EH	See Table 2-16
MSR_U_PMON_CTR0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_CTR1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_EVNT_SEL	
06_2EH	See Table 2-16
MSR_U_PMON_EVNTSEL0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_EVNTSEL1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_GLOBAL_CTRL	
06_2EH	See Table 2-16
MSR_U_PMON_GLOBAL_OVF_CTRL	
06_2EH	See Table 2-16
MSR_U_PMON_GLOBAL_STATUS	
06_2EH	See Table 2-16
MSR_U_PMON_UCLK_FIXED_CTL	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_UCLK_FIXED_CTR	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U2L_ESCR0	
0FH	See Table 2-45
MSR_U2L_ESCR1	
0FH	See Table 2-45
MSR_UNC_ARB_PERFCTR0	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_ARB_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_ARB_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_ARB_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFCTR0	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFEVTSEL1	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR0	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR1	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR3	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNC_CBO_4_PERFEVTSEL0	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL1	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_CONFIG	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_FIXED_CTR	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_FIXED_CTRL	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_GLOBAL_CTRL	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_GLOBAL_STATUS	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNCORE_ADDR_OPCODE_MATCH	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_FIXED_CTR_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_FIXED_CTR0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNCORE_PERFEVTSEL0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL3	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC3	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
06_2EH	See Table 2-16
MSR_UNCORE_PMC6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PRMRR_BASE	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_UNCORE_PRMRR_MASK	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_UNCORE_PRMRR_PHYS_BASE	
06_8EH, 06_9EH	See Table 2-40
MSR_UNCORE_PRMRR_PHYS_MASK	
06_8EH, 06_9EH	See Table 2-40
MSR_W_PMON_BOX_CTRL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
MSR_W_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_W_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_W_PMON_CTR0	
06_2EH	See Table 2-16
MSR_W_PMON_CTR1	
06_2EH	See Table 2-16
MSR_W_PMON_CTR2	
06_2EH	See Table 2-16
MSR_W_PMON_CTR3	
06_2EH	See Table 2-16
MSR_W_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_W_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_W_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_W_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_W_PMON_FIXED_CTR	
06_2EH	See Table 2-16
MSR_W_PMON_FIXED_CTR_CTL	
06_2EH	See Table 2-16
MSR_WEIGHTED_CORE_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MTRRfix16K_80000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix16K_A0000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_C0000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_C8000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_D0000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MTRRfix4K_D8000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_E0000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_E8000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_F0000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix4K_F8000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRfix64K_00000	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase0	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase1	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase2	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase3	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase4	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase5	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase6	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysBase7	
06_0EH	See Table 2-48
P6 Family	See Table 2-50

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MTRRphysMask0	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask1	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask2	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask3	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask4	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask5	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask6	
06_0EH	See Table 2-48
P6 Family	See Table 2-50
MTRRphysMask7	
06_0EH	See Table 2-48
P6 Family	See Table 2-50