

# Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

---

May 2018

**Notice:** The Intel<sup>®</sup> 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: 252046-059



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

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

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

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

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

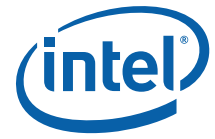
This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

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

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

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

Copyright © 1997-2018, Intel Corporation. All Rights Reserved.



# Contents

---

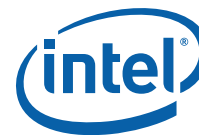
Revision History . . . . .	4
Preface . . . . .	7
Summary Tables of Changes . . . . .	8
Documentation Changes. . . . .	9



# Revision History

---

Revision	Description	Date
-001	<ul style="list-style-type: none"><li>Initial release</li></ul>	November 2002
-002	<ul style="list-style-type: none"><li>Added 1-10 Documentation Changes.</li><li>Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual</li></ul>	December 2002
-003	<ul style="list-style-type: none"><li>Added 9 -17 Documentation Changes.</li><li>Removed Documentation Change #6 - References to bits Gen and Len Deleted.</li><li>Removed Documentation Change #4 - VIF Information Added to CLI Discussion</li></ul>	February 2003
-004	<ul style="list-style-type: none"><li>Removed Documentation changes 1-17.</li><li>Added Documentation changes 1-24.</li></ul>	June 2003
-005	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-24.</li><li>Added Documentation Changes 1-15.</li></ul>	September 2003
-006	<ul style="list-style-type: none"><li>Added Documentation Changes 16- 34.</li></ul>	November 2003
-007	<ul style="list-style-type: none"><li>Updated Documentation changes 14, 16, 17, and 28.</li><li>Added Documentation Changes 35-45.</li></ul>	January 2004
-008	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-45.</li><li>Added Documentation Changes 1-5.</li></ul>	March 2004
-009	<ul style="list-style-type: none"><li>Added Documentation Changes 7-27.</li></ul>	May 2004
-010	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1.</li></ul>	August 2004
-011	<ul style="list-style-type: none"><li>Added Documentation Changes 2-28.</li></ul>	November 2004
-012	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28.</li><li>Added Documentation Changes 1-16.</li></ul>	March 2005
-013	<ul style="list-style-type: none"><li>Updated title.</li><li>There are no Documentation Changes for this revision of the document.</li></ul>	July 2005
-014	<ul style="list-style-type: none"><li>Added Documentation Changes 1-21.</li></ul>	September 2005
-015	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-21.</li><li>Added Documentation Changes 1-20.</li></ul>	March 9, 2006
-016	<ul style="list-style-type: none"><li>Added Documentation changes 21-23.</li></ul>	March 27, 2006
-017	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-23.</li><li>Added Documentation Changes 1-36.</li></ul>	September 2006
-018	<ul style="list-style-type: none"><li>Added Documentation Changes 37-42.</li></ul>	October 2006
-019	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-42.</li><li>Added Documentation Changes 1-19.</li></ul>	March 2007
-020	<ul style="list-style-type: none"><li>Added Documentation Changes 20-27.</li></ul>	May 2007
-021	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1-6</li></ul>	November 2007
-022	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-6</li></ul>	August 2008
-023	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-21</li></ul>	March 2009



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



Revision	Description	Date
-048	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-19</li><li>Add Documentation Changes 1-33</li></ul>	September 2015
-049	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-33</li></ul>	December 2015
-050	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-9</li></ul>	April 2016
-051	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-9</li><li>Add Documentation Changes 1-20</li></ul>	June 2016
-052	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-20</li><li>Add Documentation Changes 1-22</li></ul>	September 2016
-053	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-22</li><li>Add Documentation Changes 1-26</li></ul>	December 2016
-054	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-26</li><li>Add Documentation Changes 1-20</li></ul>	March 2017
-055	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-20</li><li>Add Documentation Changes 1-28</li></ul>	July 2017
-056	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28</li><li>Add Documentation Changes 1-18</li></ul>	October 2017
-057	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-18</li><li>Add Documentation Changes 1-29</li></ul>	December 2017
-058	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-29</li><li>Add Documentation Changes 1-17</li></ul>	March 2018
-059	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-17</li><li>Add Documentation Changes 1-24</li></ul>	May 2018

§

# Preface

---

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

## Affected Documents

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

## Nomenclature

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

# Summary Tables of Changes

---

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

## Codes Used in Summary Tables

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

## Documentation Changes

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 6, Volume 1
2	Updates to Chapter 13, Volume 1
3	Updates to Chapter 16, Volume 1
4	Updates to Chapter 3, Volume 2A
5	Updates to Chapter 4, Volume 2B
6	Updates to Appendix A, Volume 2D
7	Updates to Chapter 2, Volume 3A
8	Updates to Chapter 3, Volume 3A
9	Updates to Chapter 6, Volume 3A
10	Updates to Chapter 7, Volume 3A
11	Updates to Chapter 17, Volume 3B
12	Updates to Chapter 18, Volume 3B
13	Updates to Chapter 24, Volume 3B
14	Updates to Chapter 25, Volume 3C
15	Updates to Chapter 26, Volume 3C
16	Updates to Chapter 27, Volume 3C
17	Updates to Chapter 34, Volume 3C
18	Updates to Chapter 35, Volume 3C
19	Updates to Chapter 37, Volume 3D
20	Updates to Chapter 38, Volume 3D
21	Updates to Chapter 39, Volume 3D
22	Updates to Chapter 40, Volume 3D
23	Updates to Chapter 42, Volume 3D
24	Updates to Appendix C, Volume 3D



# *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 6, Volume 1

Change bars show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
Change to this chapter: Update to Table 6-1 "Exceptions and Interrupts". Updates to Section 6.4.4 "INT n, INTO, INT3, INT1, and BOUND Instructions".

# CHAPTER 6

## PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS

---

This chapter describes the facilities in the Intel 64 and IA-32 architectures for executing calls to procedures or subroutines. It also describes how interrupts and exceptions are handled from the perspective of an application programmer.

### 6.1 PROCEDURE CALL TYPES

The processor supports procedure calls in the following two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

Both of these procedure call mechanisms use the procedure stack, commonly referred to simply as “the stack,” to save the state of the calling procedure, pass parameters to the called procedure, and store local variables for the currently executing procedure.

The processor’s facilities for handling interrupts and exceptions are similar to those used by the CALL and RET instructions.

### 6.2 STACKS

The stack (see Figure 6-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. When using the flat memory model, the stack can be located anywhere in the linear address space for the program. A stack can be up to 4 GBytes long, the maximum size of a segment.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory.

When a system sets up many stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

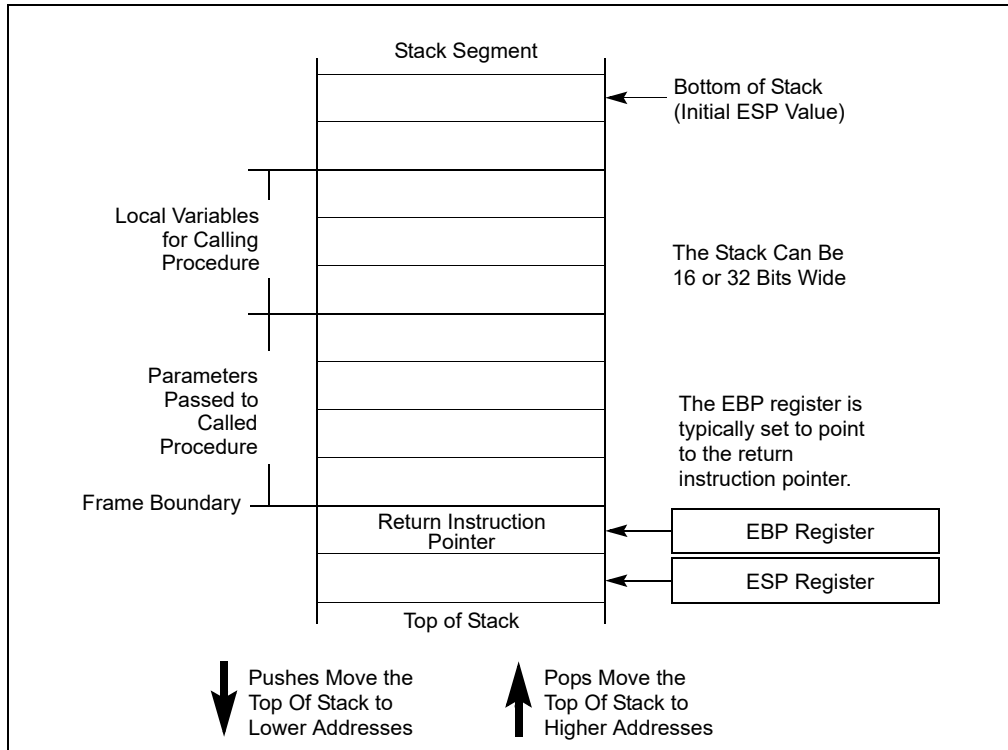


Figure 6-1. Stack Structure

The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

### 6.2.1 Setting Up a Stack

To set a stack and establish it as the current stack, the program or operating system/executive must do the following:

1. Establish a stack segment.
2. Load the segment selector for the stack segment into the SS register using a MOV, POP, or LSS instruction.
3. Load the stack pointer for the stack into the ESP register using a MOV, POP, or LSS instruction. The LSS instruction can be used to load the SS and ESP registers in one operation.

See "Segment Descriptors" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on how to set up a segment descriptor and segment limits for a stack segment.

### 6.2.2 Stack Alignment

The stack pointer for a stack segment should be aligned on 16-bit (word) or 32-bit (double-word) boundaries, depending on the width of the stack segment. The D flag in the segment descriptor for the current code segment sets the stack-segment width (see "Segment Descriptors" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). The PUSH and POP instructions use the D flag to determine how much to decrement or increment the stack pointer on a push or pop operation, respectively. When the stack width is 16 bits, the stack pointer is incremented or decremented in 16-bit increments; when the width is 32 bits, the stack pointer is incremented or decremented in 32-bit increments. Pushing a 16-bit value onto a 32-bit wide stack can result in stack misaligned (that is, the stack pointer is not aligned on a double-

word boundary). One exception to this rule is when the contents of a segment register (a 16-bit segment selector) are pushed onto a 32-bit wide stack. Here, the processor automatically aligns the stack pointer to the next 32-bit boundary.

The processor does not check stack pointer alignment. It is the responsibility of the programs, tasks, and system procedures running on the processor to maintain proper alignment of stack pointers. Misaligning a stack pointer can cause serious performance degradation and in some instances program failures.

### 6.2.3 Address-Size Attributes for Stack Accesses

Instructions that use the stack implicitly (such as the PUSH and POP instructions) have two address-size attributes each of either 16 or 32 bits. This is because they always have the implicit address of the top of the stack, and they may also have an explicit memory address (for example, PUSH Array1[EBX]). The attribute of the explicit address is determined by the D flag of the current code segment and the presence or absence of the 67H address-size prefix.

The address-size attribute of the top of the stack determines whether SP or ESP is used for the stack access. Stack operations with an address-size attribute of 16 use the 16-bit SP stack pointer register and can use a maximum stack address of FFFFH; stack operations with an address-size attribute of 32 bits use the 32-bit ESP register and can use a maximum address of FFFFFFFFH. The default address-size attribute for data segments used as stacks is controlled by the B flag of the segment's descriptor. When this flag is clear, the default address-size attribute is 16; when the flag is set, the address-size attribute is 32.

### 6.2.4 Procedure Linking Information

The processor provides two pointers for linking of procedures: the stack-frame base pointer and the return instruction pointer. When used in conjunction with a standard software procedure-call technique, these pointers permit reliable and coherent linking of procedures.

#### 6.2.4.1 Stack-Frame Base Pointer

The stack is typically divided into frames. Each stack frame can then contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack-frame base pointer (contained in the EBP register) identifies a fixed reference point within the stack frame for the called procedure. To use the stack-frame base pointer, the called procedure typically copies the contents of the ESP register into the EBP register prior to pushing any local variables on the stack. The stack-frame base pointer then permits easy access to data structures passed on the stack, to the return instruction pointer, and to local variables added to the stack by the called procedure.

Like the ESP register, the EBP register automatically points to an address in the current stack segment (that is, the segment specified by the current contents of the SS register).

#### 6.2.4.2 Return Instruction Pointer

Prior to branching to the first instruction of the called procedure, the CALL instruction pushes the address in the EIP register onto the current stack. This address is then called the return-instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the RET instruction pops the return-instruction pointer from the stack back into the EIP register. Execution of the calling procedure then resumes.

The processor does not keep track of the location of the return-instruction pointer. It is thus up to the programmer to insure that stack pointer is pointing to the return-instruction pointer on the stack, prior to issuing a RET instruction. A common way to reset the stack pointer to the point to the return-instruction pointer is to move the contents of the EBP register into the ESP register. If the EBP register is loaded with the stack pointer immediately following a procedure call, it should point to the return instruction pointer on the stack.

The processor does not require that the return instruction pointer point back to the calling procedure. Prior to executing the RET instruction, the return instruction pointer can be manipulated in software to point to any address

in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well defined code entry points.

## 6.2.5 Stack Behavior in 64-Bit Mode

In 64-bit mode, address calculations that reference SS segments are treated as if the segment base is zero. Fields (base, limit, and attribute) in segment descriptor registers are ignored. SS DPL is modified such that it is always equal to CPL. This will be true even if it is the only field in the SS descriptor that is modified.

Registers E(SP), E(IP) and E(BP) are promoted to 64-bits and are re-named RSP, RIP, and RBP respectively. Some forms of segment load instructions are invalid (for example, LDS, POP ES).

PUSH/POP instructions increment/decrement the stack using a 64-bit width. When the contents of a segment register is pushed onto 64-bit stack, the pointer is automatically aligned to 64 bits (as with a stack that has a 32-bit width).

## 6.3 CALLING PROCEDURES USING CALL AND RET

The CALL instruction allows control transfers to procedures within the current code segment (near call) and in a different code segment (far call). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task. See "CALL—Call Procedure" in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for a detailed description of the CALL instruction.

The RET instruction also allows near and far returns to match the near and far versions of the CALL instruction. In addition, the RET instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument (*n*) to the RET instruction. See "RET—Return from Procedure" in Chapter 4, "Instruction Set Reference, M-U," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a detailed description of the RET instruction.

### 6.3.1 Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 6-2):

1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
3. Resumes execution of the calling procedure.

### 6.3.2 Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 6-2):

1. Pushes the current value of the CS register on the stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. If the RET instruction has an optional  $n$  argument, increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

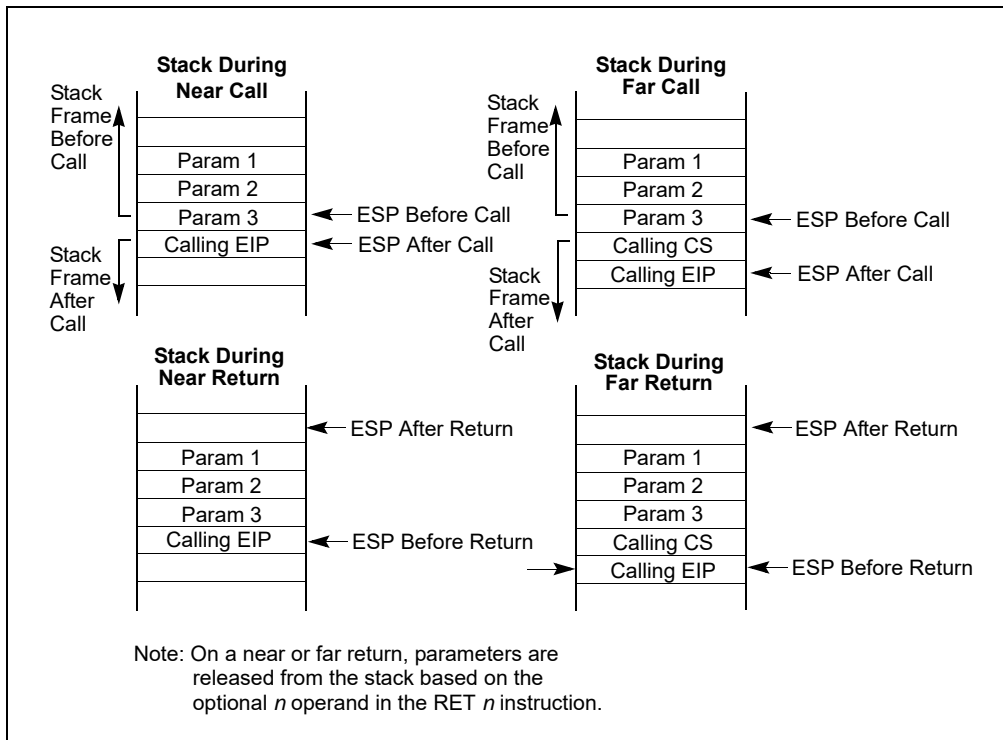


Figure 6-2. Stack on Near and Far Calls

### 6.3.3 Parameter Passing

Parameters can be passed between procedures in any of three ways: through general-purpose registers, in an argument list, or on the stack.

#### 6.3.3.1 Passing Parameters Through the General-Purpose Registers

The processor does not save the state of the general-purpose registers on procedure calls. A calling procedure can thus pass up to six parameters to the called procedure by copying the parameters into any of these registers (except the ESP and EBP registers) prior to executing the CALL instruction. The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

#### 6.3.3.2 Passing Parameters on the Stack

To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure. Here, it is useful to use the stack-frame base pointer (in the EBP register) to make a frame boundary for easy access to the parameters.

The stack can also be used to pass parameters back from the called procedure to the calling procedure.



### 6.3.3.3 Passing Parameters in an Argument List

An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory. A pointer to the argument list can then be passed to the called procedure through a general-purpose register or the stack. Parameters can also be passed back to the calling procedure in this same manner.

### 6.3.4 Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments.

The PUSHA and POPA instructions facilitate saving and restoring the contents of the general-purpose registers. PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSH instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSH instruction (except the ESP value) from the stack to their respective registers.

If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former values before executing a return to the calling procedure.

If a calling procedure needs to maintain the state of the EFLAGS register, it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

### 6.3.5 Calls to Other Privilege Levels

The IA-32 architecture’s protection mechanism recognizes four privilege levels, numbered from 0 to 3, where a greater number mean less privilege. The reason to use privilege levels is to improve the reliability of operating systems. For example, Figure 6-3 shows how privilege levels can be interpreted as rings of protection.

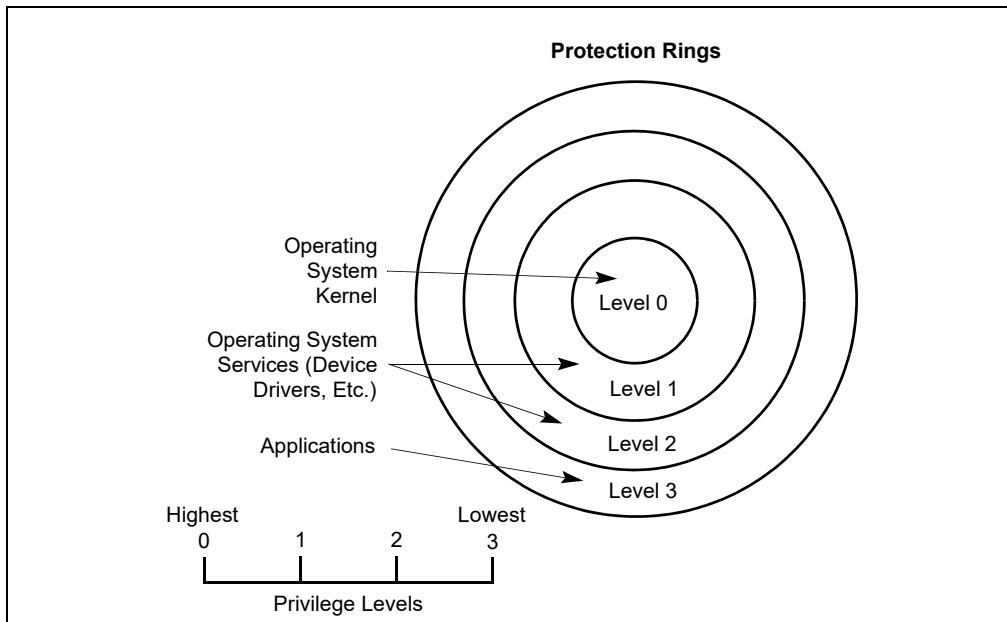


Figure 6-3. Protection Rings

In this example, the highest privilege level 0 (at the center of the diagram) is used for segments that contain the most critical code modules in the system, usually the kernel of an operating system. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software.

Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called a **gate**. Attempts to access higher privilege segments without going through a protection gate and without having sufficient access rights causes a general-protection exception (**#GP**) to be generated.

If an operating system or executive uses this multilevel protection mechanism, a call to a procedure that is in a more privileged protection level than the calling procedure is handled in a similar manner as a far call (see Section 6.3.2, "Far CALL and RET Operation"). The differences are as follows:

- The segment selector provided in the CALL instruction references a special data structure called a **call gate descriptor**. Among other things, the call gate descriptor provides the following:
  - access rights information
  - the segment selector for the code segment of the called procedure
  - an offset into the code segment (that is, the instruction pointer for the called procedure)
- The processor switches to a new stack to execute the called procedure. Each privilege level has its own stack. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and 0 stacks are stored in a system segment called the task state segment (TSS).

The use of a call gate and the TSS during a stack switch are transparent to the calling procedure, except when a general-protection exception is raised.

### 6.3.6 CALL and RET Operation Between Privilege Levels

When making a call to a more privileged protection level, the processor does the following (see Figure 6-4):

1. Performs an access rights check (privilege check).
2. Temporarily saves (internally) the current contents of the SS, ESP, CS, and EIP registers.

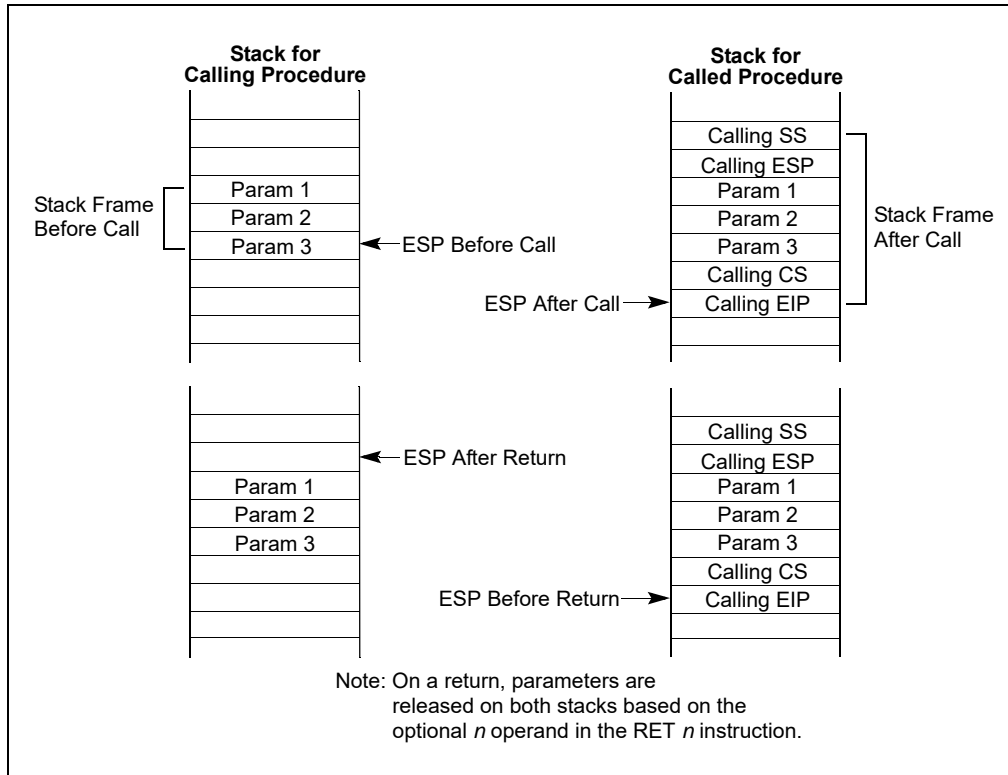


Figure 6-4. Stack Switch on a Call to a Different Privilege Level

3. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
4. Pushes the temporarily saved SS and ESP values for the calling procedure's stack onto the new stack.
5. Copies the parameters from the calling procedure's stack to the new stack. A value in the call gate descriptor determines how many parameters to copy to the new stack.
6. Pushes the temporarily saved CS and EIP values for the calling procedure to the new stack.
7. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.
8. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
3. If the RET instruction has an optional  $n$  argument, increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET  $n$  instruction must be used to release the parameters from both stacks. Here, the  $n$  operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by  $n$  for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. If the RET instruction has an optional  $n$  argument, increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

See Chapter 5, “Protection,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for detailed information on calls to privileged levels and the call gate descriptor.

### 6.3.7 Branch Functions in 64-Bit Mode

The 64-bit extensions expand branching mechanisms to accommodate branches in 64-bit linear-address space. These are:

- Near-branch semantics are redefined in 64-bit mode
- In 64-bit mode and compatibility mode, 64-bit call-gate descriptors for far calls are available

In 64-bit mode, the operand size for all near branches (CALL, RET, JCC, JCXZ, JMP, and LOOP) is forced to 64 bits. These instructions update the 64-bit RIP without the need for a REX operand-size prefix.

The following aspects of near branches are controlled by the effective operand size:

- Truncation of the size of the instruction pointer
- Size of a stack pop or push, due to a CALL or RET
- Size of a stack-pointer increment or decrement, due to a CALL or RET
- Indirect-branch operand size

In 64-bit mode, all of the above actions are forced to 64 bits regardless of operand size prefixes (operand size prefixes are silently ignored). However, the displacement field for relative branches is still limited to 32 bits and the address size for near branches is not forced in 64-bit mode.

Address sizes affect the size of RCX used for JCXZ and LOOP; they also impact the address calculation for memory indirect branches. Such addresses are 64 bits by default; but they can be overridden to 32 bits by an address size prefix.

Software typically uses far branches to change privilege levels. The legacy IA-32 architecture provides the call-gate mechanism to allow software to branch from one privilege level to another, although call gates can also be used for branches that do not change privilege levels. When call gates are used, the selector portion of the direct or indirect pointer references a gate descriptor (the offset in the instruction is ignored). The offset to the destination’s code segment is taken from the call-gate descriptor.

64-bit mode redefines the type value of a 32-bit call-gate descriptor type to a 64-bit call gate descriptor and expands the size of the 64-bit descriptor to hold a 64-bit offset. The 64-bit mode call-gate descriptor allows far branches that reference any location in the supported linear-address space. These call gates also hold the target code selector (CS), allowing changes to privilege level and default size as a result of the gate transition.

Because immediates are generally specified up to 32 bits, the only way to specify a full 64-bit absolute RIP in 64-bit mode is with an indirect branch. For this reason, direct far branches are eliminated from the instruction set in 64-bit mode.

64-bit mode also expands the semantics of the SYSENTER and SYSEXIT instructions so that the instructions operate within a 64-bit memory space. The mode also introduces two new instructions: SYSCALL and SYSRET (which are valid only in 64-bit mode). For details, see “SYSENTER—Fast System Call,” “SYSEXIT—Fast Return from Fast System Call,” “SYSCALL—Fast System Call,” and “SYSRET—Return From Fast System Call” in Chapter 4, “Instruction Set Reference, M-U,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

## 6.4 INTERRUPTS AND EXCEPTIONS

The processor provides two mechanisms for interrupting program execution, interrupts and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The IA-32 architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.

The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive through assembly-language calls. The remainder of this section gives a brief overview of the processor’s interrupt and exception handling mechanism. See Chapter 6, “Interrupt and Exception Handling,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a description of this mechanism.

The IA-32 Architecture defines 18 predefined interrupts and exceptions and 224 user defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a vector. Table 6-1 lists the interrupts and exceptions with entries in the IDT and their respective vectors. Vectors 0 through 8, 10 through 14, and 16 through 19 are the predefined interrupts and exceptions; vectors 32 through 255 are for software-defined interrupts, which are for either software interrupts or maskable hardware interrupts.

Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See Chapter 6, “Interrupt and Exception Handling,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about the interrupts and exceptions.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

### 6.4.1 Call and Return Operation for Interrupt or Exception Handling Procedures

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 6.3.6, “CALL and RET Operation Between Privilege Levels”). Here, the vector references one of two kinds of gates in the IDT: an interrupt gate or a trap gate. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information
- The segment selector for the code segment that contains the handler procedure
- An offset into the code segment to the first instruction of the handler procedure

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

**Table 6-1. Exceptions and Interrupts**

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.

**Table 6-1. Exceptions and Interrupts (Contd.)**

Vector	Mnemonic	Description	Source
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. <sup>1</sup>
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>2</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>3</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>4</sup>
20	#VE	Virtualization Exception	EPT violations <sup>5</sup>
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

**NOTES:**

1. IA-32 processors after the Intel386 processor do not generate this exception.
2. This exception was introduced in the Intel486 processor.
3. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
4. This exception was introduced in the Pentium III processor.
5. This exception can occur only on processors that support the 1-setting of the "EPT-violation #VE" VM-execution control.

If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler's privilege level.

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 6-5):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure.

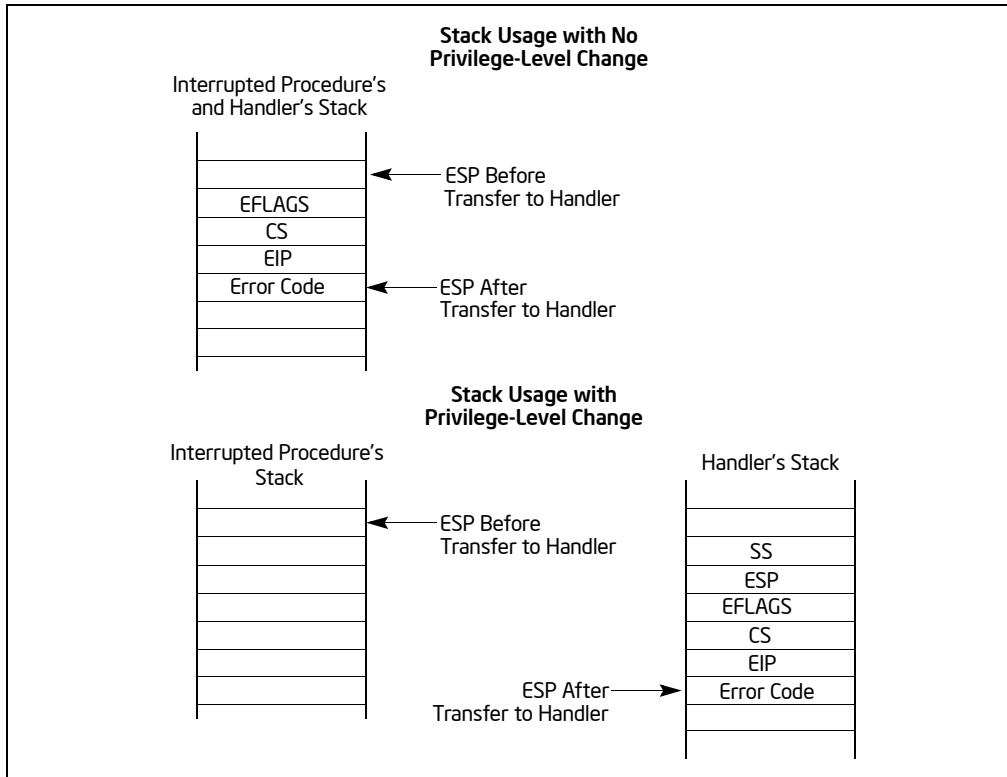


Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately.
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.

2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

## 6.4.2 Calls to Interrupt or Exception Handler Tasks

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks.

The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 6, “Interrupt and Exception Handling,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information on handling interrupts and exceptions through handler tasks.

## 6.4.3 Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures.

The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction.

See Chapter 20, “8086 Emulation,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for more information on handling interrupts and exceptions in real-address mode.

## 6.4.4 INT *n*, INTO, INT3, INT1, and BOUND Instructions

The INT *n*, INTO, INT3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction (opcode CD) uses a vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction (opcode CE) explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT3 instruction (opcode CC) explicitly calls the breakpoint exception (#BP) handler. Similarly, the INT1 instruction (opcode F1) explicitly calls the debug exception (#DB) handler.<sup>1</sup>

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays

1. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.



and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT  $n$  instruction with an argument of 5 (the vector of the bounds-check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

### 6.4.5 Handling Floating-Point Exceptions

When operating on individual or packed floating-point values, the IA-32 architecture supports a set of six floating-point exceptions. These exceptions can be generated during operations performed by the x87 FPU instructions or by SSE/SSE2/SSE3 instructions. When an x87 FPU instruction (including the FISTTP instruction in SSE3) generates one or more of these exceptions, it in turn generates floating-point error exception (#MF); when an SSE/SSE2/SSE3 instruction generates a floating-point exception, it in turn generates SIMD floating-point exception (#XM).

See the following sections for further descriptions of the floating-point exceptions, how they are generated, and how they are handled:

- Section 4.9.1, "Floating-Point Exception Conditions," and Section 4.9.3, "Typical Actions of a Floating-Point Exception Handler"
- Section 8.4, "x87 FPU Floating-Point Exception Handling," and Section 8.5, "x87 FPU Floating-Point Exception Conditions"
- Section 11.5.1, "SIMD Floating-Point Exceptions"
- Interrupt Behavior

### 6.4.6 Interrupt and Exception Behavior in 64-Bit Mode

64-bit extensions expand the legacy IA-32 interrupt-processing and exception-processing mechanism to allow support for 64-bit operating systems and applications. Changes include:

- All interrupt handlers pointed to by the IDT are 64-bit code (does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits. The processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy environments, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism.
- The alignment of interrupt stack frame is different.

## 6.5 PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES

The IA-32 architecture supports an alternate method of performing procedure calls with the ENTER (enter procedure) and LEAVE (leave procedure) instructions. These instructions automatically create and release, respectively, stack frames for called procedures. The stack frames have predefined spaces for local variables and the necessary pointers to allow coherent returns from called procedures. They also allow scope rules to be implemented so that procedures can access their own local variables and some number of other variables located in other stack frames.

ENTER and LEAVE offer two benefits:

- They provide machine-language support for implementing block-structured languages, such as C and Pascal.
- They simplify procedure entry and exit in compiler-generated code.

## 6.5.1 ENTER Instruction

The ENTER instruction creates a stack frame compatible with the scope rules typically used in block-structured languages. In block-structured languages, the scope of a procedure is the set of variables to which it has access. The rules for scope vary among languages. They may be based on the nesting of procedures, the division of the program into separately compiled files, or some other modularization scheme.

ENTER has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage for the procedure being called. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables. The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in a hierarchy of procedure calls. The lexical level is unrelated to either the protection privilege level or to the I/O privilege level of the currently running program or task.

ENTER, in the following example, allocates 2 Kbytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure:

```
ENTER 2048,3
```

The lexical nesting level determines the number of stack frame pointers to copy into the new stack frame from the preceding frame. A stack frame pointer is a doubleword used to access the variables of a procedure. The set of stack frame pointers used by a procedure to access the variables of other procedures is called the display. The first doubleword in the display is a pointer to the previous stack frame. This pointer is used by a LEAVE instruction to undo the effect of an ENTER instruction by discarding the current stack frame.

After the ENTER instruction creates the display for a procedure, it allocates the dynamic local variables for the procedure by decrementing the contents of the ESP register by the number of bytes specified in the first parameter. This new value in the ESP register serves as the initial top-of-stack for all PUSH and POP operations within the procedure.

To allow a procedure to address its display, the ENTER instruction leaves the EBP register pointing to the first doubleword in the display. Because stacks grow down, this is actually the doubleword with the highest address in the display. Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

The following pseudo code shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.

```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0
  THEN
    DO (LEVEL - 1) times
      EBP ← EBP - 4;
      PUSH Pointer(EBP); (* doubleword pointed to by EBP *)
    OD;
  PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP - STORAGE;
```

The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

A procedure that calls another procedure at a lower lexical level gives the called procedure access to the variables of the caller. The ENTER instruction provides this access by placing a pointer to the calling procedure's stack frame in the display.

A procedure that calls another procedure at the same lexical level should not give access to its variables. In this case, the ENTER instruction copies only that part of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure's stack frame.

The ENTER instruction treats a re-entrant procedure as a call to a procedure at the same lexical level. In this case, each succeeding iteration of the re-entrant procedure can address only its own variables and the variables of the procedures within which it is nested. A re-entrant procedure always can address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, the ENTER instruction makes certain that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 6-6).

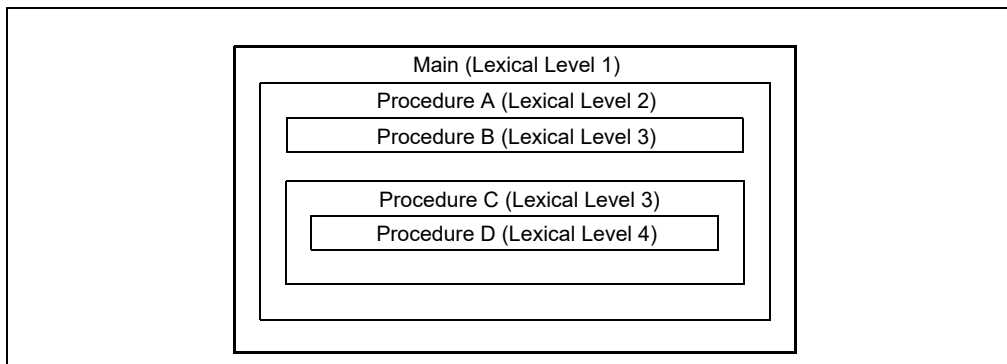


Figure 6-6. Nested Procedures

Block-structured languages can use the lexical levels defined by ENTER to control access to the variables of nested procedures. In Figure 6-6, for example, if procedure A calls procedure B which, in turn, calls procedure C, then procedure C will have access to the variables of the MAIN procedure and procedure A, but not those of procedure B because they are at the same lexical level. The following definition describes the access to variables for the nested procedures in Figure 6-6.

1. MAIN has variables at fixed locations.
2. Procedure A can access only the variables of MAIN.
3. Procedure B can access only the variables of procedure A and MAIN. Procedure B cannot access the variables of procedure C or procedure D.
4. Procedure C can access only the variables of procedure A and MAIN. Procedure C cannot access the variables of procedure B or procedure D.
5. Procedure D can access the variables of procedure C, procedure A, and MAIN. Procedure D cannot access the variables of procedure B.

In Figure 6-7, an ENTER instruction at the beginning of the MAIN procedure creates three doublewords of dynamic storage for MAIN, but copies no pointers from other stack frames. The first doubleword in the display holds a copy of the last value in the EBP register before the ENTER instruction was executed. The second doubleword holds a copy of the contents of the EBP register following the ENTER instruction. After the instruction is executed, the EBP register points to the first doubleword pushed on the stack, and the ESP register points to the last doubleword in the stack frame.

When MAIN calls procedure A, the ENTER instruction creates a new display (see Figure 6-8). The first doubleword is the last value held in MAIN's EBP register. The second doubleword is a pointer to MAIN's stack frame which is copied from the second doubleword in MAIN's display. This happens to be another copy of the last value held in MAIN's EBP register. Procedure A can access variables in MAIN because MAIN is at level 1.

Therefore the base address for the dynamic storage used in MAIN is the current address in the EBP register, plus four bytes to account for the saved contents of MAIN's EBP register. All dynamic variables for MAIN are at fixed, positive offsets from this value.

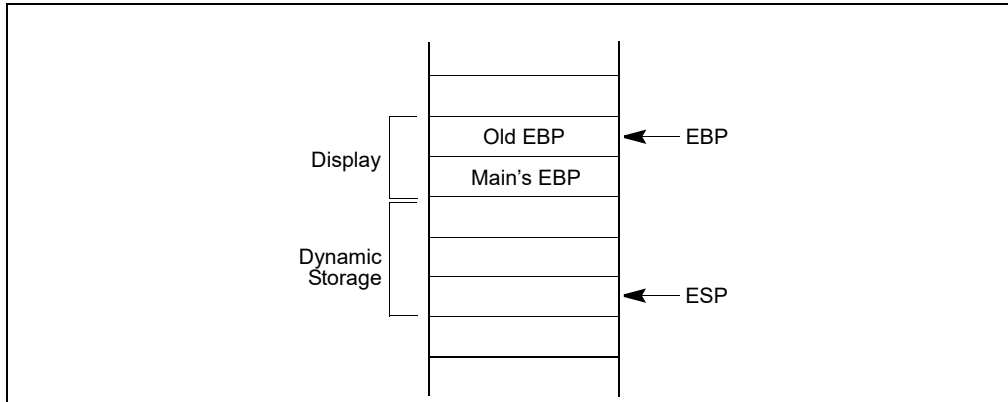


Figure 6-7. Stack Frame After Entering the MAIN Procedure

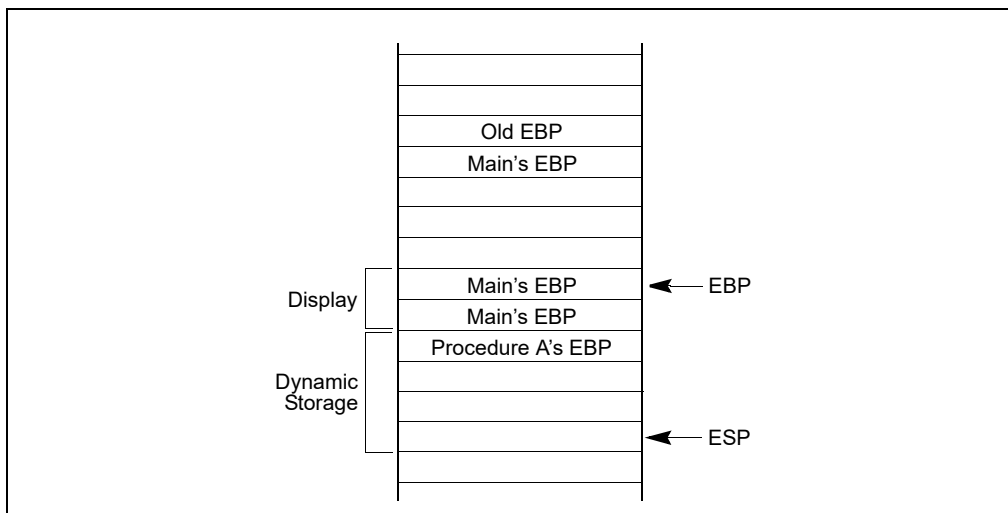


Figure 6-8. Stack Frame After Entering Procedure A

When procedure A calls procedure B, the ENTER instruction creates a new display (see Figure 6-9). The first doubleword holds a copy of the last value in procedure A's EBP register. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. Procedure B can access variables in procedure A and MAIN by using the stack frame pointers in its display.

When procedure B calls procedure C, the ENTER instruction creates a new display for procedure C (see Figure 6-10). The first doubleword holds a copy of the last value in procedure B's EBP register. This is used by the LEAVE instruction to restore procedure B's stack frame. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. If procedure C were at the next deeper lexical level from procedure B, a fourth doubleword would be copied, which would be the stack frame pointer to procedure B's local variables.

Note that procedure B and procedure C are at the same level, so procedure C is not intended to access procedure B's variables. This does not mean that procedure C is completely isolated from procedure B; procedure C is called by procedure B, so the pointer to the returning stack frame is a pointer to procedure B's stack frame. In addition, procedure B can pass parameters to procedure C either on the stack or through variables global to both procedures (that is, variables in the scope of both procedures).

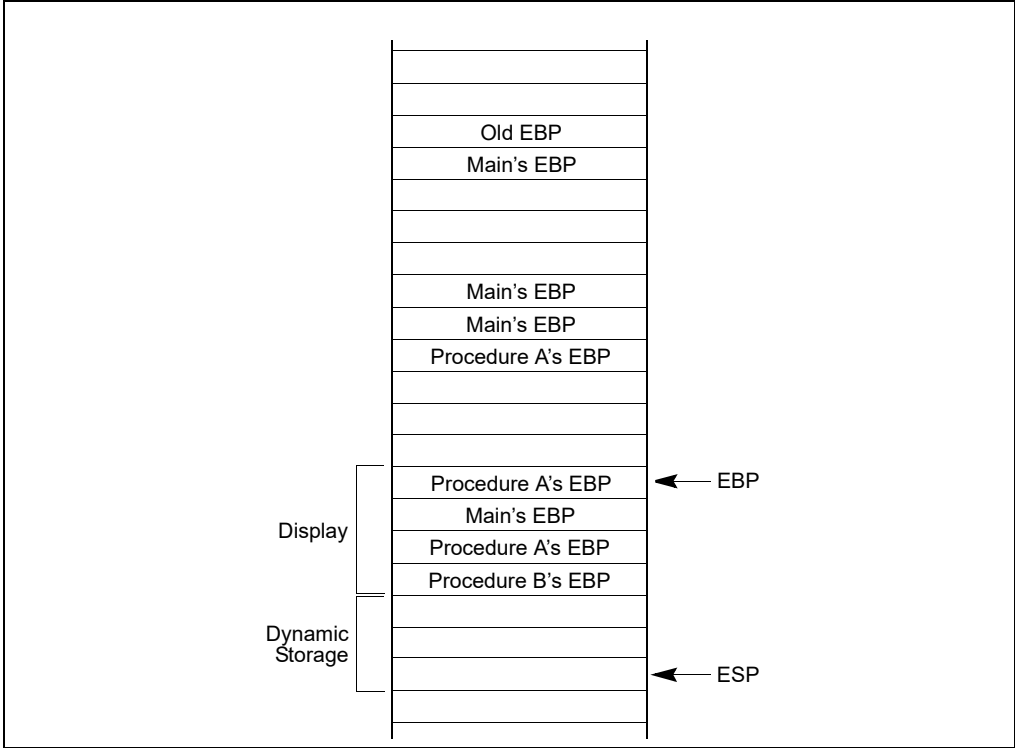


Figure 6-9. Stack Frame After Entering Procedure B

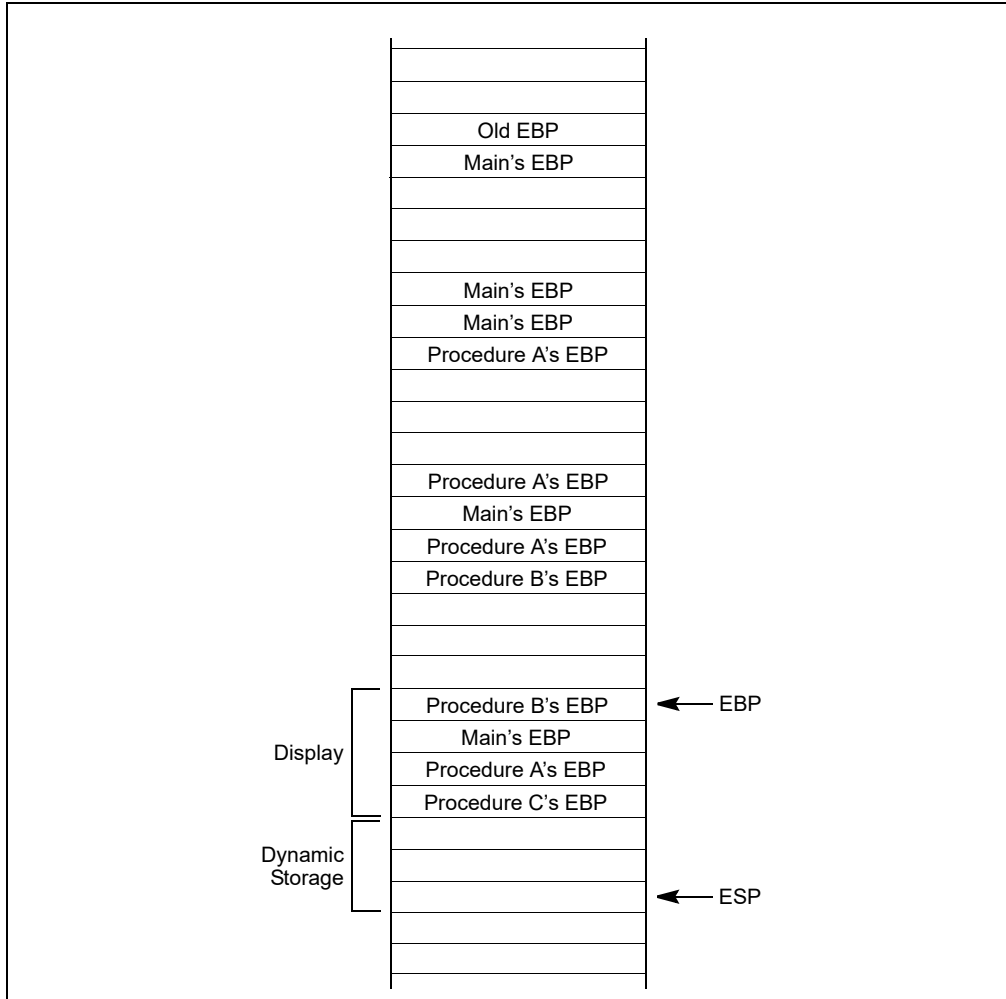


Figure 6-10. Stack Frame After Entering Procedure C

### 6.5.2 LEAVE Instruction

The LEAVE instruction, which does not have any operands, reverses the action of the previous ENTER instruction. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then it restores the old value of the EBP register from the stack. This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.



## 2. Updates to Chapter 13, Volume 1

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

-----  
Change to this chapter: Minor typo corrections.



## CHAPTER 13

# MANAGING STATE USING THE XSAVE FEATURE SET

---

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

The **XSAVE feature set** comprises eight instructions. XGETBV and XSETBV allow software to read and write the extended control register XCR0, which controls the operation of the XSAVE feature set. XSAVE, XSAVEOPT, XSAVEC, and XSAVES are four instructions that save processor state to memory; XRSTOR and XRSTORS are corresponding instructions that load processor state from memory. XGETBV, XSAVE, XSAVEOPT, XSAVEC, and XRSTOR can be executed at any privilege level; XSETBV, XSAVES, and XRSTORS can be executed only if CPL = 0. In addition to XCR0, the XSAVES and XRSTORS instructions are controlled also by the IA32\_XSS MSR (index DA0H).

The XSAVE feature set organizes the state that manages into **state components**. Operation of the instructions is based on **state-component bitmaps** that have the same format as XCR0 and as the IA32\_XSS MSR: each bit corresponds to a state component. Section 13.1 discusses these state components and bitmaps in more detail.

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

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

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

## 13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

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

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

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

- State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H (ZMM\_Hi256 state).
- State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (Hi16\_ZMM state).
- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (PT state).
- Bit 9 corresponds to the state component used for the protection-key feature’s register PKRU (PKRU state). See Section 13.5.7.
- Bit 13 corresponds to the state component used for an MSR used to control hardware duty cycling (HDC state). See Section 13.5.8.

Bits in the ranges 62:14 and 12:10 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state component is defined within bits 62:11, additional sub-sections are updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

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

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

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

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

The IA32\_XSS MSR (index DA0H) contains a state-component bitmap that specifies the supervisor state components that software has enabled XSAVES and XRSTORS to manage (XSAVE, XSAVEC, XSAVEOPT, and XRSTOR cannot manage supervisor state components). If the bit corresponding to a state component is clear in the IA32\_XSS MSR, XSAVES and XRSTORS will not operate on that state component, regardless of the value of the instruction mask.

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

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

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

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

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

- CPUID.1:ECX.XSAVE[bit 26] enumerates general support for the XSAVE feature set:
  - If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).
  - If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV.<sup>1</sup> Further enumeration is provided through CPUID function 0DH.
- CPUR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.
- CPUID function 0DH enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:
  - CPUID function 0DH, sub-function 0.
    - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).  
If  $EAX[i] = 1$  (for  $1 < i < 32$ ) or  $EDX[i-32] = 1$  (for  $32 \leq i < 63$ ), sub-function  $i$  enumerates details for state component  $i$  (see below).
    - ECX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.
    - EBX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.
  - CPUID function 0DH, sub-function 1.
    - EAX[0] enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If  $EAX[0] = 0$ , execution of XSAVEOPT causes an invalid-opcode exception (#UD).
    - EAX[1] enumerates support for **compaction extensions** to the XSAVE feature set. The following are supported if this bit is 1:
      - The compacted format of the extended region of XSAVE areas (see Section 13.4.3).
      - The XSAVEC instruction. If  $EAX[1] = 0$ , execution of XSAVEC causes a #UD.
      - Execution of the compacted form of XRSTOR (see Section 13.8).
    - EAX[2] enumerates support for execution of XGETBV with  $ECX = 1$ . This allows software to determine the state of the init optimization. See Section 13.6.
    - EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32\_XSS MSR. If  $EAX[3] = 0$ , execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32\_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32\_XSS) will also set EAX[1] (the compaction extensions).
    - EAX[31:4] are reserved.
    - EBX enumerates the size (in bytes) required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32\_XSS.
    - EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32\_XSS MSR if and only if the corresponding bit is set in this bitmap.

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

**NOTE**

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

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

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

**13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES**

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

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

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

$\text{XCR0}[1]$  is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set  $\text{XCR0}[1]$ .

- $\text{XCR0}[2]$  is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if  $\text{XCR0}[2] = 1$ . In addition, software can execute AVX instructions only if  $\text{CR4.OSXSAVE} = \text{XCR0}[2] = 1$ . Otherwise, any execution of an AVX instruction causes an invalid-opcode exception (#UD).

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

As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears  $\text{XCR0}[2]$ .

However, clearing  $\text{XCR0}[2]$  while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State

Components” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[4:3] are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if XCR0[4:3] = 11b. In addition, software can execute MPX instructions only if CR4.OSXSAVE = 1 and XCR0[4:3] = 11b. Otherwise, any execution of an MPX instruction causes an invalid-opcode exception (#UD).<sup>1</sup>

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

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

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

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

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

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

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

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

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

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

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

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
  - If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.

---

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

2. Bit 8 and bit 13 correspond to supervisor state components. Since bits can be set in XCR0 only for user state components, those bits of XCR0 must be 0.

- If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.
2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX state and software can execute MPX instructions. If XCR0[7:5] is 111b, the XSAVE feature set can be used to manage AVX-512 state and software can execute AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

The IA32\_XSS MSR (with MSR index DA0H) is zero coming out of RESET. If CR4.OSXSAVE = 1, CPUID.(EAX=0DH,ECX=1):EAX[3] = 1, and CPL = 0, executing the WRMSR instruction with ECX = DA0H writes the 64-bit value in EDX:EAX to the IA32\_XSS MSR (EAX is written to IA32\_XSS[31:0] and EDX to IA32\_XSS[63:32]). The following items provide details regarding individual bits in the IA32\_XSS MSR:

- IA32\_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32\_XSS[8] = 1. The value of IA32\_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32\_XSS[8] = 0).
- IA32\_XSS[13] is associated with HDC state (see Section 13.5.8). Software can use XSAVES and XRSTORS to manage HDC state only if IA32\_XSS[13] = 1. The value of IA32\_XSS[13] does not determine whether software can use hardware duty cycling (the feature can be used even if IA32\_XSS[13] = 0).
- IA32\_XSS[63:14], IA32\_XSS[12:9] and IA32\_XSS[7:0] are reserved.<sup>1</sup> Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32\_XSS MSR is 0 coming out of RESET.

There is no mechanism by which software operating with CPL > 0 can discover the value of the IA32\_XSS MSR.

## 13.4 XSAVE AREA

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

Every XSAVE area has the following format:

- The **legacy region**. The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It is used to manage the state components for x87 state and SSE state. The legacy region is described in more detail in Section 13.4.1.
- The **XSAVE header**. The XSAVE header of an XSAVE area comprises the 64 bytes starting at an offset of 512 bytes from the area's base address. The XSAVE header is described in more detail in Section 13.4.2.
- The **extended region**. The extended region of an XSAVE area starts at an offset of 576 bytes from the area's base address. It is used to manage the state components other than those for x87 state and SSE state. The extended region is described in more detail in Section 13.4.3. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 and IA32\_XSS (see Section 13.3).

### 13.4.1 Legacy Region of an XSAVE Area

The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It has the same format as the FXSAVE area (see Section 10.5.1). The XSAVE feature set uses the legacy area for x87 state (state

---

1. Bit 9 and bits 7:0 correspond to user state components. Since bits can be set in the IA32\_XSS MSR only for supervisor state components, those bits of the MSR must be 0.

component 0) and SSE state (state component 1). Table 13-1 illustrates the format of the first 416 bytes of the legacy region of an XSAVE area.

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

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
FIP[63:48] or reserved	FCS or FIP[47:32]	FIP[31:0]		FOP	Rsvd.	FTW	FSW	FCW	<b>0</b>
MXCSR_MASK		MXCSR		FDP[63:48] or reserved	FDS or FDP[47:32]		FDP[31:0]		<b>16</b>
Reserved			Reserved			ST0/MM0			<b>32</b>
Reserved			Reserved			ST1/MM1			<b>48</b>
Reserved			Reserved			ST2/MM2			<b>64</b>
Reserved			Reserved			ST3/MM3			<b>80</b>
Reserved			Reserved			ST4/MM4			<b>96</b>
Reserved			Reserved			ST5/MM5			<b>112</b>
Reserved			Reserved			ST6/MM6			<b>128</b>
Reserved			Reserved			ST7/MM7			<b>144</b>
			XMM0						<b>160</b>
			XMM1						<b>176</b>
			XMM2						<b>192</b>
			XMM3						<b>208</b>
			XMM4						<b>224</b>
			XMM5						<b>240</b>
			XMM6						<b>256</b>
			XMM7						<b>272</b>
			XMM8						<b>288</b>
			XMM9						<b>304</b>
			XMM10						<b>320</b>
			XMM11						<b>336</b>
			XMM12						<b>352</b>
			XMM13						<b>368</b>
			XMM14						<b>384</b>
			XMM15						<b>400</b>

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

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

### 13.4.2 XSAVE Header

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

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called **XSTATE\_BV**. It identifies the state components in the XSAVE area.

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

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

### 13.4.3 Extended Region of an XSAVE Area

The extended region of an XSAVE area starts at byte offset 576 from the area’s base address. The size of the extended region is determined by which state components the processor supports and which bits have been set in **XCR0 | IA32\_XSS** (see Section 13.3).

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

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

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

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

## 13.5 XSAVE-MANAGED STATE

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



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

### 13.5.1 x87 State

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

- Bytes 1:0, 3:2, 7:6. These are used for the x87 FPU Control Word (FCW), the x87 FPU Status Word (FSW), and the x87 FPU Opcode (FOP), respectively.
- Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  - For each  $j$ ,  $0 \leq j \leq 7$ , XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 0 into bit  $j$  of byte 4 if x87 FPU data register  $ST_j$  has an empty tag; otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 1 into bit  $j$  of byte 4.
  - For each  $j$ ,  $0 \leq j \leq 7$ , XRSTOR and XRSTORS establish the tag value for x87 FPU data register  $ST_j$  as follows. If bit  $j$  of byte 4 is 0, the tag for  $ST_j$  in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for  $ST_j$  based on the value being loaded into that register (see below).
- Bytes 15:8 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    - If  $CPUID.(EAX=07H,ECX=0H):EBX[\text{bit } 13] = 0$ , bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FCS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H, and XRSTOR and XRSTORS ignore them.
    - Bytes 15:14 are not used.
  - If the instruction has a REX prefix with  $REX.W = 1$ , bytes 15:8 are used for the full 64 bits of FIP.
- Bytes 23:16 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).
    - If  $CPUID.(EAX=07H,ECX=0H):EBX[\text{bit } 13] = 0$ , bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
    - Bytes 23:22 are not used.
  - If the instruction has a REX prefix with  $REX.W = 1$ , bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers  $ST_0$ – $ST_7$  ( $MM_0$ – $MM_7$ ). Each of the 8 registers is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

x87 state is XSAVE-managed but the x87 FPU feature is not XSAVE-enabled. The XSAVE feature set can operate on x87 state only if the feature set is enabled ( $CR_4.OSXSAVE = 1$ ).<sup>1</sup> Software can otherwise use x87 state even if the XSAVE feature set is not enabled.

### 13.5.2 SSE State

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

1. The processor ensures that  $XCRO[0]$  is always 1.

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.<sup>1</sup>
- Bytes 31:28 are used for the MXCSR\_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

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

### 13.5.3 AVX State

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

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

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

The XSAVE feature set partitions YMM0\_H–YMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are used for YMM0\_H–YMM7\_H. Bytes 255:128 are used for YMM8\_H–YMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8\_H–YMM15\_H. See Section 13.13. In general, bytes 16/+15:16*i* are used for YMM*i*\_H (for 0 ≤ *i* ≤ 15).

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

### 13.5.4 MPX State

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

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

- **BNDREGS state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREGS state (when the

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

standard format of the extended region is used). CPUID.(EAX=0DH,ECX=3):EAX enumerates the size (in bytes) required for BNDREGS state. The BNDREGS section is used for the 4 128-bit bound registers BND0–BND3, with bytes  $16i+15:16i$  being used for BND*i*.

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

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

### 13.5.5 AVX-512 State

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

- The opmask registers, collectively called **opmask state**.
- The upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H and are collectively called **ZMM\_Hi256 state**.
- The 16 512-bit registers ZMM16–ZMM31, collectively called **Hi16\_ZMM state**.

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

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

- **Opmask state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes) required for opmask state. The opmask section is used for the 8 64-bit bound registers k0–k7, with bytes  $8i+7:8i$  being used for k*i*.
- **ZMM\_Hi256 state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM\_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM\_Hi256 state.  
The XSAVE feature set partitions ZMM0\_H–ZMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM\_Hi256-state section are used for ZMM0\_H–ZMM7\_H. Bytes 511:256 are used for ZMM8\_H–ZMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8\_H–ZMM15\_H. See Section 13.13. In general, bytes  $32i+31:32i$  are used for ZMM*i*\_H (for  $0 \leq i \leq 15$ ).
- **Hi16\_ZMM state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16\_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16\_ZMM state.  
The XSAVE feature set accesses Hi16\_ZMM state only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify the Hi16\_ZMM section; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM16–ZMM31. See Section 13.13. In general, bytes  $64(i-16)+63:64(i-16)$  are used for ZMM*i* (for  $16 \leq i \leq 31$ ).

All three components of AVX-512 state are XSAVE-managed and the AVX-512 feature is XSAVE-enabled. The XSAVE feature set can operate on AVX-512 state only if the feature set is enabled (CR4.OSXSAVE = 1) and has

been configured to manage AVX-512 state (XCR0[7:5] = 111b). AVX-512 instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX-512 state.

### 13.5.6 PT State

The register state used by Intel Processor Trace (PT state) comprises the following 9 MSRs: IA32\_RTIT\_CTL, IA32\_RTIT\_OUTPUT\_BASE, IA32\_RTIT\_OUTPUT\_MASK\_PTRS, IA32\_RTIT\_STATUS, IA32\_RTIT\_CR3\_MATCH, IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B, IA32\_RTIT\_ADDR1\_A, and IA32\_RTIT\_ADDR1\_B.<sup>1</sup>

As noted in Section 13.1, the XSAVE feature set manages PT state as supervisor state component 8. Thus, PT state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=8):EAX enumerates the size (in bytes) required for PT state. The MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32\_RTIT\_CTL is at byte offset 0, IA32\_RTIT\_OUTPUT\_BASE at byte offset 8, etc. Any locations in the state component at or beyond byte offset 72 are reserved.

PT state is XSAVE-managed but Intel Processor Trace is not XSAVE-enabled. The XSAVE feature set can operate on PT state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PT state (IA32\_XSS[8] = 1). Software can otherwise use Intel Processor Trace and access its MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PT state.

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

- If XSAVES saves PT state, the instruction clears IA32\_RTIT\_CTL.TraceEn (bit 0) after saving the value of the IA32\_RTIT\_CTL MSR and before saving any other PT state. If XSAVES causes a fault or a VM exit, it restores IA32\_RTIT\_CTL.TraceEn to its original value.
- If XSAVES saves PT state, the instruction saves zeroes in the reserved portions of the state component.
- If XRSTORS would restore (or initialize) PT state and IA32\_RTIT\_CTL.TraceEn = 1, the instruction causes a general-protection exception (#GP) before modifying PT state.
- If XRSTORS causes an exception or a VM exit, it does so before any modification to IA32\_RTIT\_CTL.TraceEn (even if it has loaded other PT state).

### 13.5.7 PKRU State

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

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

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

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

---

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

### 13.5.8 HDC State

The register state used by hardware duty cycling (HDC state) comprises the IA32\_PM\_CTL1 MSR.

As noted in Section 13.1, the XSAVE feature set manages HDC state as supervisor state component 13. Thus, HDC state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=13):EAX enumerates the size (in bytes) required for PT state. The IA32\_PM\_CTL1 MSR is allocated 8 bytes at byte offset 0 in the state component.

HDC state is XSAVE-managed but hardware duty cycling is not XSAVE-enabled. The XSAVE feature set can operate on HDC state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HDC state (IA32\_XSS[13] = 1). Software can otherwise use hardware duty cycle and access the IA32\_PM\_CTL1 MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HDC state.

## 13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

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

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

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

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

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

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

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

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

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

- **x87 state.** x87 state is in its initial configuration if the following all hold: FCW is 037FH; FSW is 0000H; FTW is FFFFH; FCS and FDS are each 0000H; FIP and FDP are each 00000000\_00000000H; each of ST0–ST7 is 0000\_00000000\_00000000H.

- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM15 is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM7 is 0. XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update XMM8–XMM15. (See Section 13.13.)
- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of YMM0\_H–YMM15\_H is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of YMM0\_H–YMM7\_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update YMM8\_H–YMM15\_H. (See Section 13.13.)
- **BNDREGS state.** BNDREGS state is in its initial configuration if the value of each of BND0–BND3 is 0.
- **BNDCSR state.** BNDCSR state is in its initial configuration if BNDCFGU and BNDCSR each has value 0.
- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers k0–k7 is 0.
- **ZMM\_Hi256 state.** In 64-bit mode, ZMM\_Hi256 state is in its initial configuration if each of ZMM0\_H–ZMM15\_H is 0. Outside 64-bit mode, ZMM\_Hi256 state is in its initial configuration if each of ZMM0\_H–ZMM7\_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM8\_H–ZMM15\_H. (See Section 13.13.)
- **Hi16\_ZMM state.** In 64-bit mode, Hi16\_ZMM state is in its initial configuration if each of ZMM16–ZMM31 is 0. Outside 64-bit mode, Hi16\_ZMM state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM31–ZMM31. (See Section 13.13.)
- **PT state.** PT state is in its initial configuration if each of the 9 MSR is 0.
- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.
- **HDC state.** HDC state is in its initial configuration if the value of the IA32\_PM\_CTL1 MSR is 0.

## 13.7 OPERATION OF XSAVE

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

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

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

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

- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is not changed.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$ . Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component  $i$  is in its initial configuration,  $XINUSE[i]$  may be either 0 or 1, and  $XSTATE\_BV[i]$  may be written with either 0 or 1.  
 XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. Thus,  $XSTATE\_BV[1]$  may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If state component  $i$  is not in its initial configuration,  $XINUSE[i] = 1$  and  $XSTATE\_BV[i]$  is written with 1.  
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVE instruction does not write any part of the XSAVE header other than the XSTATE\_BV field; in particular, it does not write to the XCOMP\_BV field.

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

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

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

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

## 13.8 OPERATION OF XRSTOR

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

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

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

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

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

### 13.8.1 Standard Form of XRSTOR

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

- The XSTATE\_BV field of the XSAVE header sets a bit that is not set in XCR0.
- Bytes 23:8 of the XSAVE header are not all 0 (this implies that all bits in XCOMP\_BV are 0).<sup>3</sup>

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

- If XSTATE\_BV[ $i$ ] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR

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

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

2. If the processor does not support the compacted form of XRSTOR, it may execute the standard form of XRSTOR without first reading the XCOMP\_BV field. A processor supports the compacted form of XRSTOR only if it enumerates CPUID.(EAX=0DH,ECX=1):EAX[1] as 1.

3. Bytes 63:24 of the XSAVE header are also reserved. Software should ensure that bytes 63:16 of the XSAVE header are all 0 in any XSAVE area. (Bytes 15:8 should also be 0 if the XSAVE area is to be used on a processor that does not support the compaction extensions to the XSAVE feature set.)

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

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

### 13.8.2 Compacted Form of XRSTOR

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

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

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

- If XSTATE\_BV[ $i$ ] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

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

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

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

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

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

### 13.8.3 XRSTOR and the Init and Modified Optimizations

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

- The processor updates its tracking for the init optimization as follows:
  - If RFBM[ $i$ ] = 0, XINUSE[ $i$ ] is not changed.
  - If RFBM[ $i$ ] = 1 and XSTATE\_BV[ $i$ ] = 0, state component  $i$  may be tracked as init; XINUSE[ $i$ ] may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the init optimization for state component  $i$ ; a processor that does not do so implicitly maintains XINUSE[ $i$ ] = 1 at all times.)
  - If RFBM[ $i$ ] = 1 and XSTATE\_BV[ $i$ ] = 1, state component  $i$  is tracked as not init; XINUSE[ $i$ ] is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTOR execution for future interaction with the XSAVEOPT and XSAVES instructions (see Section 13.9 and Section 13.11) as follows:
  - If RFBM[ $i$ ] = 0, state component  $i$  is tracked as modified; XMODIFIED[ $i$ ] is set to 1.

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and XSTATE\_BV[ $i$ ] is 1, then XCOMP\_BV[ $i$ ] is also 1.



- If  $RFBM[i] = 1$ , state component  $i$  may be tracked as unmodified;  $XMODIFIED[i]$  may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the modified optimization for state component  $i$ ; a processor that does not do so implicitly maintains  $XMODIFIED[i] = 1$  at all times.)
- $XRSTOR\_INFO$  is set to the 4-tuple  $\langle w, x, y, z \rangle$ , where  $w$  is the CPL (0);  $x$  is 1 if the logical processor is in VMX non-root operation and 0 otherwise;  $y$  is the linear address of the XSAVE area; and  $z$  is  $XCOMP\_BV$ . In particular, the standard form of  $XRSTOR$  always sets  $z$  to all zeroes, while the compacted form of  $XRSTORS$  never does so (because it sets at least bit 63 to 1).

## 13.9 OPERATION OF XSAVEOPT

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

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

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

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

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

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

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

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

The  $XSAVEOPT$  instruction does not write any part of the XSAVE header other than the  $XSTATE\_BV$  field; in particular, it does not write to the  $XCOMP\_BV$  field.

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

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

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

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

- **Init optimization.**

If  $XINUSE[i] = 0$ , state component  $i$  is not saved to the XSAVE area (even if  $RFBM[i] = 1$ ). (See below for exceptions made for MXCSR.)

- **Modified optimization.**

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

- $w = \text{CPL}$ ;
- $x = 1$  if and only if the logical processor is in VMX non-root operation;
- $y$  is the linear address of the XSAVE area being used by XSAVEOPT; and
- $z$  is 00000000\_00000000H. (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of XRSTOR used the compacted form, or if an execution of XRSTORS followed the last execution of XRSTOR.)

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

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

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

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

## 13.10 OPERATION OF XSAVEC

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

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

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

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

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE\_BV field of the XSAVE header (see Section 13.4.2), setting  $XSTATE\_BV[i]$  ( $0 \leq i \leq 63$ ) as follows:<sup>2</sup>

- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is written as 0.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$  (see below for an exception made for  $XSTATE\_BV[1]$ ). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component  $i$  is in its initial configuration,  $XSTATE\_BV[i]$  may be written with either 0 or 1.

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

2. Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does **not** read the XSTATE\_BV field of the XSAVE header.

- If state component  $i$  is not in its initial configuration, `XSTATE_BV[ $i$ ]` is written with 1.

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

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

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

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

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

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If `XINUSE[ $i$ ] = 0`, state component  $i$  is not saved to the XSAVE area (even if `RFBM[ $i$ ] = 1`). However, if `RFBM[1] = 1` and MXCSR does not have the value 1F80H, XSAVEC writes saves all of state component 1 (SSE — including the XMM registers) even if `XINUSE[1] = 0`. Unlike the XSAVE instruction, `RFBM[2]` does not determine whether XSAVEC saves MXCSR and `MXCSR_MASK`.

## 13.11 OPERATION OF XSAVES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 13.12 OPERATION OF XRSTORS

The operation of XRSTORS is similar to that of XRSTOR. Three main differences are (1) XRSTORS can be executed only if  $\text{CPL} = 0$ ; (2) XRSTORS can operate on the state components whose bits are set in  $\text{XCR0} \mid \text{IA32\_XSS}$  and can thus operate on supervisor state components; and (3) XRSTORS has only a compacted form (no standard form; see Section 13.8). See Section 13.2 for details of how to determine whether XRSTORS is supported.

The XRSTORS instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**.  $\text{EDX:EAX} \& (\text{XCR0} \mid \text{IA32\_XSS})$  (the logical AND the instruction mask with the logical OR of XCR0 and IA32\_XSS) is the **requested-feature bitmap (RFBM)** of the state components to be restored.

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

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

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

- $\text{XCOMP\_BV}[63] = 0$ .
- XCOMP\_BV sets a bit in the range 62:0 that is not set in  $\text{XCR0} \mid \text{IA32\_XSS}$ .
- XSTATE\_BV sets a bit (including bit 63) that is not set in XCOMP\_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

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

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

- If  $XSTATE\_BV[i] = 0$ , the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If  $XSTATE\_BV[1] = 0$ , XRSTORS initializes MXCSR to 1F80H.  
State component  $i$  is set to its initial configuration as indicated above if  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 0$  — even if  $XCOMP\_BV[i] = 0$ . This is true for all values of  $i$ , including 0 (x87 state) and 1 (SSE state).
- If  $XSTATE\_BV[i] = 1$ , the state component is loaded with data from the XSAVE area.<sup>1</sup> See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XRSTORS. See Section 13.13 for details regarding faults caused by memory accesses.

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

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

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

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

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

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

## 13.13 MEMORY ACCESSSES BY THE XSAVE FEATURE SET

Each instruction in the XSAVE feature set operates on a set of XSAVE-managed state components. The specific set of components on which an instruction operates is determined by the values of XCR0, the IA32\_XSS MSR, EDX:EAX, and (for XRSTOR and XRSTORS) the XSAVE header.

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

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and  $XSTATE\_BV[i]$  is 1, then  $XCOMP\_BV[i]$  is also 1.

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

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

For example, an execution of XSAVE outside 64-bit mode may incur a page fault if paging does not map as read/write the section of the XSAVE area containing state component 7 (Hi16\_ZMM state) — despite the fact that state component 7 can be accessed only in 64-bit mode.

### 3. Updates to Chapter 16, Volume 1

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

-----  
Change to this chapter: Updates to Section 16.3.8.1 "Instruction Based Considerations".

### 16.1 OVERVIEW

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

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

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

### 16.2 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

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

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

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

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

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

Intel TSX provides two software interfaces for programmers.

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



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

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

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

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

## 16.2.1 HLE Software Interface

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

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

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

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

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

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

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

## 16.2.2 RTM Software Interface

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

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

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

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

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

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

## 16.3 INTEL® TSX APPLICATION PROGRAMMING MODEL

### 16.3.1 Detection of Transactional Synchronization Support

#### 16.3.1.1 Detection of HLE Support

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

#### 16.3.1.2 Detection of RTM Support

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

#### 16.3.1.3 Detection of XTEST Instruction

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

## 16.3.2 Querying Transactional Execution Status

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

## 16.3.3 Requirements for HLE Locks

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

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

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

## 16.3.4 Transactional Nesting

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

### 16.3.4.1 HLE Nesting and Elision

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

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

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

### 16.3.4.2 RTM Nesting

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

### 16.3.4.3 Nesting HLE and RTM

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

## 16.3.5 RTM Abort Status Definition

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

**Table 16-1. RTM Abort Status Definition**

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

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

### 16.3.6 RTM Memory Ordering

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

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

## 16.3.7 RTM-Enabled Debugger Support

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

Advanced debugging of RTM transactional regions is enabled if bit 11 of DR7 and bit 15 of the IA32\_DEBUGCTL MSR are both 1. In this case, any RTM transactional abort due to a #DB or #BP causes execution to roll back to just before the XBEGIN instruction (EAX is restored to the value it had before XBEGIN) and then delivers a #DB. (A #DB is delivered even if the transactional abort was caused by a #BP.) DR6[16] is cleared to indicate that the exception resulted from a debug or breakpoint exception inside an RTM region. See also Section 17.3.3, “Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM),” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## 16.3.8 Programming Considerations

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

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

### 16.3.8.1 Instruction Based Considerations

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

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

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

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

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

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

The following instructions will abort transactional execution on any implementation:

- XABORT
- CPUID
- PAUSE
- ENCLS
- ENCLU

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

- Operations on X87 and MMX architecture state. This includes all MMX and X87 instructions, including the FXRSTOR and FXSAVE instructions.
- Update to non-status portion of EFLAGS: CLI, STI, POPFD, POPFQ, CLAC and STAC.
- Instructions that update segment registers, debug registers and/or control registers: MOV to DS/ES/FS/GS/SS, POP DS/ES/FS/GS/SS, LDS, LES, LFS, LGS, LSS, SWAPGS, WRFSBASE, WRGSBASE, LGDT, SGDT, LIDT, SIDT, LLDT, SLDT, LTR, STR, Far CALL, Far JMP, Far RET, IRET, MOV to DRx, MOV to CR0/CR2/CR3/CR4/CR8, CLTS, and LMSW.
- Ring transitions: SYSENTER, SYSCALL, SYSEXIT, and SYSRET.
- TLB and Cacheability control: CLFLUSH, CLFLUSHOPT, CLWB, INVD, WBINVD, INVLPG, INVPCID, and memory instructions with a non-temporal hint (V/MOVNTDQA, V/MOVNTDQ, V/MOVNTI, V/MOVNTPD, V/MOVNTPS, V/MOVNTQ, V/MASKMOVQ, and V/MASKMOVDQU).
- Extended state management: XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV.
- Interrupts: INT  $n$ , INTO, INT3, INT1.
- I/O: IN, INS, REP INS, OUT, OUTS, REP OUTS and their variants.
- VMX: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, INVVPID, and VMFUNC.
- SMX: GETSEC.
- UD0, UD1, UD2, RSM, RDMSR, WRMSR, WRPKRU, HLT, MONITOR, MWAIT, and VZERoupper.

### 16.3.8.2 Runtime Considerations

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

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

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

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

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

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

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

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

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

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

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

#### 4. Updates to Chapter 3, Volume 2A

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

-----

Change to this chapter: Update to Table 3-4 "Intel 64 and IA-32 General Exceptions". Updates to INT n/INTO/INT3/INT1 instruction.



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

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

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

Vector	Name	Source	Protected Mode <sup>7</sup>	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	Reserved	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	Reserved	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection <sup>2</sup>	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	Reserved	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	Reserved	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes

**NOTES:**

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

### 3.1.1.14 Real-Address Mode Exceptions Section

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

### 3.1.1.15 Virtual-8086 Mode Exceptions Section

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

**INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT3	Z0	Valid	Valid	Generate breakpoint trap.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INTO	Z0	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	Z0	Valid	Valid	Generate debug trap.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	imm8	NA	NA	NA

**Description**

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.<sup>1</sup> Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT *n* instruction (including the INTO, INT3, and INT1 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

Each of the INT *n*, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.

even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug.)

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

**Table 3-51. Decision Table**

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

**NOTES:**

- Don't Care.
- Y Yes, action taken.
- Blank Action not taken.
- S/W Applies to INT *n*, INT3, and INTO, but not to INT1.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT *n* instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

**Operation**

The following operational description applies not only to the INT *n*, INTO, INT3, or INT1 instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction `error_code(num,idt,ext)`, where `idt` and `ext` are bit values. The pseudofunction produces an error code as follows: (1) if `idt` is 0, the error code is  $(num \& FCH) | ext$ ; (2) if `idt` is 1, the error code is  $(num \ll 3) | 2 | ext$ .

In many cases, the pseudofunction `error_code` is invoked with a pseudovalue `EXT`. The value of `EXT` depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt (`INT n`, `INT3`, or `INTO`), `EXT` is 0; otherwise (including `INT1`), `EXT` is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      ELSE
        IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
          THEN
            Consult bit n of the software interrupt redirection bit map in the TSS;
            IF bit n is clear
              THEN (* redirect interrupt to 8086 program interrupt handler *)
                Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                Push CS;
                Push IP;
                IF IOPL = 3
                  THEN IF ← 0; (* Clear interrupt flag *)
                  ELSE VIF ← 0; (* Clear virtual interrupt flag *)
                FI;
                TF ← 0; (* Clear trap flag *)
                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
              ELSE
                IF IOPL = 3
                  THEN GOTO PROTECTED-MODE;
                  ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                FI;
            FI;
          ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
            IF (IA32_EFER.LMA = 0)
              THEN (* Protected mode, or virtual-8086 mode interrupt *)
                GOTO PROTECTED-MODE;
              ELSE (* IA-32e mode interrupt *)
                GOTO IA-32e-MODE;
            FI;
          FI;
        FI;
      FI;
    FI;
  REAL-ADDRESS-MODE:
    IF ((vector_number ≪ 2) + 3) is not within IDT limit
      THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
      THEN #SS; FI;
    Push (EFLAGS[15:0]);

```

```

IF ← 0; (* Clear interrupt flag *)
TF ← 0; (* Clear trap flag *)
AC ← 0; (* Clear AC flag *)
Push(CS);
Push(IP);
(* No error codes are pushed in real-address mode*)
CS ← IDT(Descriptor (vector_number << 2), selector));
EIP ← IDT(Descriptor (vector_number << 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
IF ((vector_number << 3) + 7) is not within IDT limits
or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
  THEN #GP(error_code(vector_number,1,EXT)); FI;
  (* idt operand to error_code set because vector is used *)
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
  THEN
  IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
    THEN #GP(error_code(vector_number,1,0)); FI;
    (* idt operand to error_code set because vector is used *)
    (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
  FI;
  IF gate not present
    THEN #NP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
  IF task gate (* Specified in the selected interrupt table descriptor *)
    THEN GOTO TASK-GATE;
    ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
  FI;
END;
IA-32e-MODE:
IF INTO and CS.L = 1 (64-bit mode)
  THEN #UD;
  FI;
IF ((vector_number << 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
  THEN #GP(error_code(vector_number,1,EXT));
  (* idt operand to error_code set because vector is used *)
  FI;
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
  THEN
  IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
    THEN #GP(error_code(vector_number,1,0));
    (* idt operand to error_code set because vector is used *)
    (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
  FI;
  IF gate not present
    THEN #NP(error_code(vector_number,1,EXT));
    (* idt operand to error_code set because vector is used *)
  FI;
  GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
  Read TSS selector in task gate (IDT descriptor);

```

```

IF local/global bit is set to local or index not within GDT limits
    THEN #GP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Access TSS descriptor in GDT;
IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
    THEN #GP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF TSS not present
    THEN #NP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
SWITCH-TASKS (with nesting) to TSS;
IF interrupt caused by fault with error code
    THEN
        IF stack limit does not allow push of error code
            THEN #SS(EXT); FI;
        Push(error code);
    FI;
IF EIP not within code segment limit
    THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
Read new code-segment selector for trap or interrupt gate (IDT descriptor);
IF new code-segment selector is NULL
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
IF new code-segment selector is not within its descriptor table limits
    THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Read descriptor referenced by new code-segment selector;
IF descriptor does not indicate a code segment or new code-segment DPL > CPL
    THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new code-segment descriptor is not present,
    THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new code segment is non-conforming with DPL < CPL
    THEN
        IF VM = 0
            THEN
                GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                DPL < CPL *)
            ELSE (* VM = 1 *)
                IF new code-segment DPL ≠ 0
                    THEN #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
                GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            IF new code segment is conforming or new code-segment DPL = CPL
                THEN

```

```

        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
    ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
        #GP(error_code(new code-segment selector,0,EXT));
        (* idt operand to error_code is 0 because selector is used *)
    FI;
FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        (* Identify stack-segment selector for new privilege level in current TSS *)
        IF current TSS is 32-bit
            THEN
                TSSstackAddress ← (new code-segment DPL << 3) + 4;
                IF (TSSstackAddress + 5) > current TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 16-bit *)
                TSSstackAddress ← (new code-segment DPL << 2) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
            FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI;
        IF NewSS index is not within its descriptor-table limits
        or NewSS RPL ≠ new code-segment DPL
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Read new stack-segment descriptor for NewSS in GDT or LDT;
        IF new stack-segment DPL ≠ new code-segment DPL
        or new stack-segment Type does not indicate writable data segment
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF NewSS is not present
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IA-32e mode *)
        IF IDT-gate IST = 0
            THEN TSSstackAddress ← (new code-segment DPL << 3) + 4;
            ELSE TSSstackAddress ← (IDT gate IST << 3) + 28;
        FI;
        IF (TSSstackAddress + 7) > current TSS limit
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
    FI;
    IF IDT gate is 32-bit

```

```

    THEN
        IF new stack does not have room for 24 bytes (error code pushed)
        or 20 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    FI
ELSE
    IF IDT gate is 16-bit
        THEN
            IF new stack does not have room for 12 bytes (error code pushed)
            or 10 bytes (no error code pushed);
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
            FI;
        FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limits
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            ESP ← NewESP;
            SS ← NewSS; (* Segment descriptor information also loaded *)
        ELSE (* IA-32e mode *)
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFOH;
            SS ← NewSS;
        FI;
    IF IDT gate is 32-bit
        THEN
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit IDT gate *)
                    CS:RIP ← Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
                FI;
            FI;
        FI;
    IF IDT gate is 32-bit
        THEN
            Push(far pointer to old stack);
            (* Old SS and ESP, 3 words padded to 4 *)
            Push(EFLAGS);
            Push(far pointer to return instruction);
            (* Old CS and EIP, 3 words padded to 4 *)
            Push(ErrorCode); (* If needed, 4 bytes *)
        ELSE
            IF IDT gate 16-bit
                THEN

```



```

        Push(far pointer to old stack);
        (* Old SS and SP, 2 words *)
        Push(EFLAGS(15:0));
        Push(far pointer to return instruction);
        (* Old CS and IP, 2 words *)
        Push(ErrorCode); (* If needed, 2 bytes *)
    ELSE (* 64-bit IDT gate *)
        Push(far pointer to old stack);
        (* Old SS and SP, each an 8-byte push *)
        Push(RFLAGS); (* 8-byte push *)
        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8-bytes *)

    FI;

FI;
CPL ← new code-segment DPL;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
    THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
TF ← 0;
VM ← 0;
RF ← 0;
NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
    THEN
        IF TSS limit < 9
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 8);
            NewESP ← 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
                NewSS ← 2 bytes loaded from (current TSS base + 4);
                NewESP ← 2 bytes loaded from (current TSS base + 2);

    FI;
IF NewSS is NULL
    THEN #TS(EXT); FI; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF IDT gate is 32-bit

```

```

THEN
    IF new stack does not have room for 40 bytes (error code pushed)
    or 36 bytes (no error code pushed)
        THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IDT gate is 16-bit)
        IF new stack does not have room for 20 bytes (error code pushed)
        or 18 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS ← Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) ← 0;
CPL ← 0;
IF IDT gate is 32-bit
    THEN
        EIP ← Gate(instruction pointer);
    ELSE (* IDT gate is 16-bit *)
        EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
    IF IDT-descriptor IST ≠ 0

```

```

THEN
    TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
    IF (TSSstackAddress + 7) > TSS limit
        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
        NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
    ELSE NewRSP ← RSP;
FI;
FI;
IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
    THEN
        IF current stack does not have room for 16 bytes (error code pushed)
            or 12 bytes (no error code pushed)
            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
                or 6 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(EXT); (* Error code contains NULL selector *)
            FI;
        FI;
    FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limit
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFF0H;
        FI;
    FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
        Push (EFLAGS);
        Push (far pointer to return instruction); (* 3 words padded to 4 *)
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
    ELSE
        IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
            THEN
                Push (FLAGS);
                Push (far pointer to return location); (* 2 words *)
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
                Push (ErrorCode); (* If any *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
                Push(far pointer to return instruction);
                (* Old CS and RIP, each an 8-byte push *)
                Push(ErrorCode); (* If needed, 8 bytes *)
                CS:RIP ← GATE(CS:RIP);
            FI;
        FI;
    FI;

```

```

(* Segment descriptor information also loaded *)
    FI;
FI;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
    THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;
END;

```

### Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the “Operation” section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task’s TSS.

### Protected Mode Exceptions

#GP(error_code)	<p>If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt, trap, or task gate is NULL.</p> <p>If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt, trap, or task gate.</p> <p>If an interrupt is generated by the INT <i>n</i>, INT3, or INTO instruction and the DPL of an interrupt, trap, or task gate is less than the CPL.</p> <p>If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(error_code)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.</p>
#NP(error_code)	<p>If code segment, interrupt gate, trap gate, task gate, or TSS is not present.</p>
#TS(error_code)	<p>If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.</p> <p>If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#UD	<p>If the LOCK prefix is used.</p>
#AC(EXT)	<p>If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.</p>

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push. If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT <i>n</i> , INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3. If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits. If the segment selector in the interrupt, trap, or task gate is NULL. If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits. If the vector selects a descriptor outside the IDT limits. If an IDT descriptor is not an interrupt, trap, or task gate. If an interrupt is generated by INT <i>n</i> , INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL. If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment. If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present. If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. If the stack segment selector in the TSS is NULL. If the stack segment for the TSS is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(error_code)	<p>If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.</p> <p>If the segment selector in the 64-bit interrupt or trap gate is NULL.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If the vector points to a gate which is in non-canonical space.</p>
#SS(error_code)	<p>If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.</p> <p>If the descriptor pointed to by the gate selector is outside the descriptor table limit.</p> <p>If the descriptor pointed to by the gate selector is in non-canonical space.</p> <p>If the descriptor pointed to by the gate selector is not a code segment.</p> <p>If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.</p> <p>If the descriptor pointed to by the gate selector has DPL &gt; CPL.</p>
#NP(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
#TS(error_code)	<p>If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).</p> <p>If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.</p>
#PF(fault-code)	If an attempt to load RSP from the TSS causes an access to non-canonical space.
#UD	If the RSP from the TSS is outside descriptor table limits.
#AC(EXT)	If a page fault occurs.
	If the LOCK prefix is used.
	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

## 5. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U*.

-----  
Changes to this chapter: Updates to MOV, POP and STI instructions.

## 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 <sup>***</sup> ,r8 <sup>***</sup>	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 <sup>***</sup> ,r/m8 <sup>***</sup>	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 <sup>**</sup>	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r16/r32/m16, Sreg <sup>**</sup>	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/r64/m16.
REX.W + 8C /r	MOV r64/m16, Sreg <sup>**</sup>	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg,r/m16 <sup>**</sup>	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64 <sup>**</sup>	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8 <sup>*</sup>	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8 <sup>*</sup>	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16 <sup>*</sup>	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32 <sup>*</sup>	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64 <sup>*</sup>	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 <sup>***</sup> ,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16 <sup>*</sup> ,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32 <sup>*</sup> ,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64 <sup>*</sup> ,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 <sup>***</sup> ,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 /O ib	MOV r/m8,imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /O ib	MOV r/m8 <sup>***</sup> ,imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /O iw	MOV r/m16,imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /O id	MOV r/m32,imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /O 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 suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an event can be delivered. See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

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 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)	<p>If the memory address is in a non-canonical form.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL = 3.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL &lt; 3 and CPL ≠ RPL.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the memory access to the descriptor table is non-canonical.</p> <p>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.</p> <p>If the SS register is being loaded and the segment pointed to is a nonwritable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>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.</p>
#SS(0)	<p>If the stack address is in a non-canonical form.</p>
#SS(selector)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

## POP—Pop a Value from the Stack

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
8F /0	POP <i>r/m16</i>	M	Valid	Valid	Pop top of stack into <i>m16</i> ; increment stack pointer.
8F /0	POP <i>r/m32</i>	M	N.E.	Valid	Pop top of stack into <i>m32</i> ; increment stack pointer.
8F /0	POP <i>r/m64</i>	M	Valid	N.E.	Pop top of stack into <i>m64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
58+ <i>rw</i>	POP <i>r16</i>	0	Valid	Valid	Pop top of stack into <i>r16</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r32</i>	0	N.E.	Valid	Pop top of stack into <i>r32</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r64</i>	0	Valid	N.E.	Pop top of stack into <i>r64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	Z0	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	Z0	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	Z0	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	Z0	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	Z0	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	Z0	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	Z0	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	Z0	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	Z0	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
0	opcode + rd (w)	NA	NA	NA
Z0	NA	NA	NA	NA

### Description

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

Address and operand sizes are determined and used as follows:

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

The address size is used only when writing to a destination operand in memory.

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

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

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

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

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

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

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

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

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

Loading the SS register with a POP instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (POP ESP) before an event can be delivered. See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

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

## Operation

```

IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
        ESP ← ESP + 2;
    FI;
  ELSE IF StackAddrSize = 64
    THEN
      IF OperandSize = 64
        THEN

```

```

        DEST ← SS:RSP; (* Copy quadword *)
        RSP ← RSP + 8;
    ELSE (* OperandSize = 16*)
        DEST ← SS:RSP; (* Copy a word *)
        RSP ← RSP + 2;
    FI;
FI;
ELSE StackAddrSize = 16
    THEN
        IF OperandSize = 16
            THEN
                DEST ← SS:SP; (* Copy a word *)
                SP ← SP + 2;
            ELSE (* OperandSize = 32 *)
                DEST ← SS:SP; (* Copy a doubleword *)
                SP ← SP + 4;
            FI;
        FI;
FI;

```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

## 64-BIT\_MODE

```

IF 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 (both RPL and CPL > DPL))
                THEN #GP(selector);
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;
FI;
IF FS, or GS is loaded with a NULL selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;

```

## PROTECTED MODE OR COMPATIBILITY MODE;

```

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 (both RPL and 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 a NULL selector
  THEN
    SegmentRegister ← segment selector;
    SegmentRegister ← segment descriptor;
  FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>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.</p> <p>If the SS register is being loaded and the segment pointed to is a non-writable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>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.</p>



#SS(0)	If the current top of stack is not within the stack segment. 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 an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit. If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment. If the 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.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

## STI—Set Interrupt Flag

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FB	STI	Z0	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. This allows the processor to respond to maskable hardware interrupts.

If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI. (The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure or subroutine. For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized. No interrupts can be recognized if an execution of CLI immediately follow such an execution of STI.) The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and nonmaskable interrupts (NMIs). However, NMIs (and system-management interrupts) may be inhibited on the instruction boundary following an execution of STI that begins with IF = 0.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-19 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

**Table 4-19. Decision Table for STI Results**

Mode	IOPL	EFLAGS.VIP	STI Result
Real-address	X <sup>1</sup>	X	IF = 1
Protected, not PVI <sup>2</sup>	≥ CPL	X	IF = 1
	< CPL	X	#GP fault
Protected, PVI <sup>3</sup>	3	X	IF = 1
	0-2	0	VIF = 1
		1	#GP fault
Virtual-8086, not VME <sup>3</sup>	3	X	IF = 1
	0-2	X	#GP fault
Virtual-8086, VME <sup>3</sup>	3	X	IF = 1
	0-2	0	VIF = 1
		1	#GP fault

#### NOTES:

1. X = This setting has no effect on instruction operation.

2. For this table, “protected mode” applies whenever CRO.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

## Operation

```

IF CRO.PE = 0 (* Executing in real-address mode *)
  THEN IF ← 1; (* Set Interrupt Flag *)
  ELSE
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
      THEN IF ← 1; (* Set Interrupt Flag *)
      ELSE
        IF VME mode OR PVI mode
          THEN
            IF EFLAGS.VIP = 0
              THEN VIF ← 1; (* Set Virtual Interrupt Flag *)
              ELSE #GP(0);
            FI;
          ELSE #GP(0);
        FI;
      FI;
    FI;
  FI;

```

## Flags Affected

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

## Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active. If CPL is greater than IOPL and EFLAGS.VIP = 1.
#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 IOPL is less than 3 and VME mode is not active. If IOPL is less than 3 and EFLAGS.VIP = 1.
#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.

## 6. Updates to Appendix A, Volume 2D

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

-----  
Change to this chapter: Updates to Table A-2 "One-byte Opcode Map".

Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

## NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

## A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined. Cells marked "Reserved-NOP" are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked "Reserved-NOP" nor depend on the current behavior of those opcodes.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, "Opcode Look-up Examples for One, Two, and Three-Byte Opcodes."

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

## A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- B The VEX.vvvv field of the VEX prefix selects a general purpose register.

- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).
- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS/RFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.
- I Immediate data: the operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.
- O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

## A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.

dq	Double-quadword, regardless of operand-size attribute.
p	32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd	128-bit or 256-bit packed double-precision floating-point data.
pi	Quadword MMX technology register (for example: mm0).
ps	128-bit or 256-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
qq	Quad-Quadword (256-bits), regardless of operand-size attribute.
s	6-byte or 10-byte pseudo-descriptor.
sd	Scalar element of a 128-bit double-precision floating data.
ss	Scalar element of a 128-bit single-precision floating data.
si	Doubleword integer register (for example: eax).
v	Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w	Word, regardless of operand-size attribute.
x	dq or qq based on the operand-size attribute.
y	Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

### A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

### A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

#### A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.

- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSH A; 06H, PUSH ES).

#### Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem\_op, and the offset of mem\_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

### A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

#### Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA405000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).



- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:0000000H, EAX, 3.

### A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3AH. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

#### Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

### A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix "v" is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

## A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

**Table A-1. Superscripts Utilized in Opcode Tables**

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction), the 0FB9H opcode (UD1 instruction), or the 0FFFH opcode (UDO instruction) when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

## A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table A-2. One-byte Opcode Map: (00H – F7H) \*

	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH ES <sup>64</sup>	POP ES <sup>64</sup>
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH SS <sup>64</sup>	POP SS <sup>64</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=ES (Prefix)	DAA <sup>64</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=SS (Prefix)	AAA <sup>64</sup>
4	INC <sup>64</sup> general register / REX <sup>64</sup> Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH <sup>d64</sup> general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>64</sup> / PUSHAD <sup>64</sup>	POPA <sup>64</sup> / POPAD <sup>64</sup>	BOUND <sup>64</sup> Gv, Ma	ARPL <sup>64</sup> Ew, Gw MOVSD <sup>64</sup> Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc <sup>64</sup> , Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1 <sup>1A</sup>			TEST		XCHG		
	Eb, lb	Ev, lz	Eb, lb <sup>64</sup>	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV							
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register							
	AL/R8L, lb	CL/R9L, lb	DL/R10L, lb	BL/R11L, lb	AH/R12L, lb	CH/R13L, lb	DH/R14L, lb	BH/R15L, lb
C	Shift Grp 2 <sup>1A</sup>		near RET <sup>64</sup> lw	near RET <sup>64</sup>	LES <sup>64</sup> Gz, Mp VEX+2byte	LDS <sup>64</sup> Gz, Mp VEX+1byte	Grp 11 <sup>1A</sup> - MOV	
	Eb, lb	Ev, lb					Eb, lb	Ev, lz
D	Shift Grp 2 <sup>1A</sup>				AAM <sup>64</sup> lb	AAD <sup>64</sup> lb	XLAT/ XLATB	
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE <sup>64</sup> / LOOPNZ <sup>64</sup> Jb	LOOPE <sup>64</sup> / LOOPZ <sup>64</sup> Jb	LOOP <sup>64</sup> Jb	Jrcxz <sup>64</sup> / Jb	IN	eAX, lb	OUT	
					AL, lb		lb, AL	lb, eAX
F	LOCK (Prefix)	INT1	REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3 <sup>1A</sup>	
							Eb	Ev

Table A-2. One-byte Opcode Map: (08H – FFH) \*

	8	9	A	B	C	D	E	F	
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH CS <sup>64</sup>	2-byte escape (Table A-3)	
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH DS <sup>64</sup>	POP DS <sup>64</sup>	
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=CS (Prefix)	DAS <sup>64</sup>	
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=DS (Prefix)	AAS <sup>64</sup>	
4	DEC <sup>64</sup> general register / REX <sup>64</sup> Prefixes								
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB	
5	POP <sup>64</sup> into general register								
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15	
6	PUSH <sup>d64</sup> lz	IMUL Gv, Ev, lz	PUSH <sup>d64</sup> lb	IMUL Gv, Ev, lb	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz	
7	Jcc <sup>64</sup> , Jb- Short displacement jump on condition								
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
8	MOV						MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				Grp 1A <sup>1A</sup> POP <sup>d64</sup> Ev	
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL <sup>i64</sup> Ap	FWAIT/ WAIT	PUSHF/D/Q <sup>d64</sup> / Fv	POPF/D/Q <sup>d64</sup> / Fv	SAHF	LAHF	
A	TEST AL, lb		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv	
B	MOV immediate word or double into word, double, or quad register								
	rAX/r8, lv	rCX/r9, lv	rDX/r10, lv	rBX/r11, lv	rSP/r12, lv	rBP/r13, lv	rSI/r14, lv	rDI/r15, lv	
C	ENTER lw, lb	LEAVE <sup>d64</sup>	far RET lw	far RET	INT3	INT lb	INTO <sup>i64</sup>	IRET/D/Q	
D	ESC (Escape to coprocessor instruction set)								
E	near CALL <sup>i64</sup> Jz	near <sup>f64</sup> Jz	JMP far <sup>i64</sup> Ap	short <sup>f64</sup> Jb	AL, DX	eAX, DX	DX, AL	DX, eAX	
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>	

NOTES:

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-3. Two-byte Opcode Map: 00H – 77H (First Byte is 0FH) \*

	pxf	0	1	2	3	4	5	6	7
0		Grp 6 <sup>1A</sup>	Grp 7 <sup>1A</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL <sup>064</sup>	CLTS	SYSRET <sup>064</sup>
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps <sup>v1</sup> Vdq, Hq, Mq vmovhlps Vdq, Hq, Uq	vmovhps <sup>v1</sup> Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpd <sup>v1</sup> Vdq, Hq, Mq	vmovhpd <sup>v1</sup> Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy, Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12 <sup>1A</sup> )	(Grp 13 <sup>1A</sup> )	(Grp 14 <sup>1A</sup> )	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper <sup>v</sup> vzeroall <sup>v</sup>
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshuffw Vx, Wx, Ib							
	F2	vpshufw Vx, Wx, Ib							

Table A-3. Two-byte Opcode Map: 08H – 7FH (First Byte is 0FH) \*

	px	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2 <sup>1B</sup>		prefetchw(/1) Ev		
1		Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )	Reserved-NOP	bndldx	bndstx	Reserved-NOP			NOP /0 Ev
	66			bndmov	bndmov				
	F3			bndcl	bndmk				
	F2			bndcu	bndcn				
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvtps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvtpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtss2ss Vss, Hss, Ey		vcvtss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtss2sd Vsd, Hsd, Ey		vcvtss2si Gy, Wsd	vcvtss2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtqd2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hs, Wss	vcvttps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtss2ss Vss, Hs, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsbpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx, Vx
	F3							vmovq Vq, Wq	vmovdqu Wx, Vx
	F2					vhaddps Vps, Hps, Wps	vhsbps Vps, Hps, Wps		

Table A-3. Two-byte Opcode Map: 80H – F7H (First Byte is 0FH) \*

	pxf	0	1	2	3	4	5	6	7
8		Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH <sup>d64</sup> FS	POP <sup>d64</sup> FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX	
		Eb, Gb	Ev, Gv					Gv, Eb	Gv, Ew
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9 <sup>1A</sup>
	66			vcmpsd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpps Vss,Hss,Wss,Ib					
	F2			vcmpsd Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvttd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

**Table A-3. Two-byte Opcode Map: 88H – FFH (First Byte is 0FH) \***

	pxf	8	9	A	B	C	D	E	F
8		Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH <sup>d64</sup> GS	POP <sup>d64</sup> GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev
B		JMPE (reserved for emulator on IPF)	Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSBX Gv, Eb	Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpqr Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmmaxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	UD0
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



**Table A-4. Three-byte Opcode Map: 00H – F7H (First Two Bytes are 0F 38H) \***

	pxf	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phadd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphadd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtp2ps <sup>v</sup> Vx, Wx, lb	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps <sup>v</sup> Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd <sup>v</sup> Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q <sup>v</sup> Vx, Hx, Wx	vpsravd <sup>v</sup> Vx, Hx, Wx	vpsrlvd/q <sup>v</sup> Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q <sup>v</sup> Vx,Hx,Wx	vgatherqd/q <sup>v</sup> Vx,Hx,Wx	vgatherdps/d <sup>v</sup> Vx,Hx,Wx	vgatherqps/d <sup>v</sup> Vx,Hx,Wx			vfmaddsub132ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd132ps/d <sup>v</sup> Vx,Hx,Wx
A	66							vfmaddsub213ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd213ps/d <sup>v</sup> Vx,Hx,Wx
B	66							vfmaddsub231ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd231ps/d <sup>v</sup> Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN <sup>v</sup> Gy, By, Ey	Grp 17 <sup>1A</sup>		BZHI <sup>v</sup> Gy, Ey, By		BEXTR <sup>v</sup> Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX <sup>v</sup> Gy, Ey, By
	F3						PEXT <sup>v</sup> Gy, By, Ey	ADOX Gy, Ey	SARX <sup>v</sup> Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP <sup>v</sup> Gy, By, Ey	MULX <sup>v</sup> By,Gy,rDX,Ey	SHRX <sup>v</sup> Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

**Table A-4. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 38H) \***

	pxf	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps <sup>v</sup> Vx,Hx,Wx	vpermilpd <sup>v</sup> Vx,Hx,Wx	vtestps <sup>v</sup> Vx, Wx	vtestpd <sup>v</sup> Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss <sup>v</sup> Vx, Wd	vbroadcastsd <sup>v</sup> Vqq, Wq	vbroadcastf128 <sup>v</sup> Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps <sup>v</sup> Vx,Hx,Mx	vmaskmovpd <sup>v</sup> Vx,Hx,Mx	vmaskmovps <sup>v</sup> Mx,Hx,Vx	vmaskmovpd <sup>v</sup> Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpmins d Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
4									
5	66	vpbroadcast <sup>v</sup> Vx, Wx	vpbroadcastq <sup>v</sup> Vx, Wx	vpbroadcasti128 <sup>v</sup> Vqq, Mdq					
6									
7	66	vpbroadcastb <sup>v</sup> Vx, Wx	vpbroadcastw <sup>v</sup> Vx, Wx						
8	66					vpmaskmovd/q <sup>v</sup> Vx,Hx,Mx		vpmaskmovd/q <sup>v</sup> Mx,Vx,Hx	
9	66	vfmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ss/d <sup>v</sup> Vx, Hx, Wx
A	66	vfmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ss/d <sup>v</sup> Vx, Hx, Wx
B	66	vfmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ss/d <sup>v</sup> Vx, Hx, Wx
C		sha1nexte Vdq,Wdq	sha1msg1 Vdq,Wdq	sha1msg2 Vdq,Wdq	sha256rmds2 Vdq,Wdq	sha256msg1 Vdq,Wdq	sha256msg2 Vdq,Wdq		
	66								
D	66				VAESIMC Vdq, Wdq	VAESEC Vdq,Hdq,Wdq	VAESENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
E									
F									
	66								
	F3								
	F2								
	66 & F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) \***

	px	0	1	2	3	4	5	6	7
0	66	vpermq <sup>v</sup> Vqq, Wqq, lb	vpermpd <sup>v</sup> Vqq, Wqq, lb	vblendd <sup>v</sup> Vx,Hx,Wx,lb		vpermilps <sup>y</sup> Vx, Wx, lb	vpermilpd <sup>y</sup> Vx, Wx, lb	vperm2f128 <sup>y</sup> Vqq,Hqq,Wqq,lb	
1	66					vpextrb Rd/Mb, Vdq, lb	vpextrw Rd/Mw, Vdq, lb	vpextrd/q Ey, Vdq, lb	vextractps Ed, Vdq, lb
2	66	vpinsrb Vdq,Hdq,Ry/Mb,lb	vinsertps Vdq,Hdq,Udq/Md,lb	vpinsrd/q Vdq,Hdq,Ey,lb					
3									
4	66	vdpps Vx,Hx,Wx,lb	vdppd Vdq,Hdq,Wdq,lb	vmpsadbw Vx,Hx,Wx,lb		vpcmlulqdd Vdq,Hdq,Wdq,lb		vperm2i128 <sup>y</sup> Vqq,Hqq,Wqq,lb	
5									
6	66	vpcmpestrm Vdq, Wdq, lb	vpcmpestri Vdq, Wdq, lb	vpcmpistrm Vdq, Wdq, lb	vpcmpistri Vdq, Wdq, lb				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX <sup>v</sup> Gy, Ey, lb							

Table A-5. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 3AH) \*

	px	8	9	A	B	C	D	E	F
0									palgnr Pq, Qq, Ib
	66	vroundps Vx, Wx, Ib	vroundpd Vx, Wx, Ib	vroundss Vss, Wss, Ib	vroundsd Vsd, Wsd, Ib	vblendps Vx, Hx, Wx, Ib	vblendpd Vx, Hx, Wx, Ib	vpblendw Vx, Hx, Wx, Ib	vpalignr Vx, Hx, Wx, Ib
1	66	vinserf128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextractf128 <sup>v</sup> Wdq, Vqq, Ib				vcvtps2ph <sup>y</sup> Wx, Vx, Ib		
2									
3	66	vinserf128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextractf128 <sup>v</sup> Wdq, Vqq, Ib						
4	66			vblendvps <sup>v</sup> Vx, Hx, Wx, Lx	vblendvpd <sup>v</sup> Vx, Hx, Wx, Lx	vpblendvb <sup>v</sup> Vx, Hx, Wx, Lx			
5									
6									
7									
8									
9									
A									
B									
C						sha1rmds4 Vdq, Wdq, Ib			
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

mod	nnn	R/M
-----	-----	-----

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

### A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

#### Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

#### Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

### A.4.2 Opcode Extension Tables

See Table A-6 below.

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)								
				000	001	010	011	100	101	110	111	
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP	
8F	1A	mem, 11B		POP								
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR	
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX	
FE	4	mem, 11B		INC Eb	DEC Eb							
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL <sup>f64</sup> Ev	far CALL Ep	near JMP <sup>f64</sup> Ev	far JMP Mp	PUSH <sup>d64</sup> Ev		
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew			
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb	
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001)  VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)					SWAPGS <sup>064</sup> (000) RDTSCP (001)	
0F BA	8	mem, 11B						BT	BTS	BTR	BTC	
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq	
			66							VMCLEAR Mq		
		11B	F3								VMXON Mq	
			F3								RDRAND Rv	RDSEED Rv
0F B9	10	mem 11B								UD1		
C6	11	mem		MOV Eb, lb								
11B										XABORT (000) lb		
C7	11	mem		MOV Ev, lz								
11B											XBEGIN (000) Jz	
0F 71	12	mem										
		11B	66			psrlw Nq, lb		psraw Nq, lb		psllw Nq, lb		
0F 72	13	11B				psrld Nq, lb		psrad Nq, lb		pslld Nq, lb		
			66			vpsrlw Hx,Ux,lb		vpsraw Hx,Ux,lb		vpsllw Hx,Ux,lb		
0F 73	14	11B				psrlq Nq, lb				psllq Nq, lb		
			66			vpsrlq Hx,Ux,lb	vpsrldq Hx,Ux,lb			vpsllq Hx,Ux,lb	vpslldq Hx,Ux,lb	

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \* (Contd.)**

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B	F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry		lfence	mfence	sfence
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2	Reserved NOP			
		11B		Reserved NOP							
VEX.0F38 F3	17	mem			BLSR <sup>v</sup> By, Ey	BLSMSK <sup>v</sup> By, Ey	BLSI <sup>v</sup> By, Ey				
		11B									

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

### A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

#### Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

#### Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

### A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

#### A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.2 Escape Opcodes with D9 as First Byte**

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	EXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.3 Escape Opcodes with DA as First Byte**

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.4 Escape Opcodes with DB as First Byte**

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.5 Escape Opcodes with DC as First Byte**

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.6 Escape Opcodes with DD as First Byte**

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.7 Escape Opcodes with DE as First Byte**

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMPP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.8 Escape Opcodes with DF As First Byte**

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

OPCODE MAP

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



This page was  
intentionally left  
blank.



## 7. Updates to Chapter 2, Volume 3A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Change to this chapter: Update to Section 2.1.4 "Interrupt and Exception Handling".

IA-32 architecture (beginning with the Intel386 processor family) provides extensive support for operating-system and system-development software. This support offers multiple modes of operation, which include:

- Real mode, protected mode, virtual 8086 mode, and system management mode. These are sometimes referred to as legacy modes.

Intel 64 architecture supports almost all the system programming facilities available in IA-32 architecture and extends them to a new operating mode (IA-32e mode) that supports a 64-bit programming environment. IA-32e mode allows software to operate in one of two sub-modes:

- 64-bit mode supports 64-bit OS and 64-bit applications
- Compatibility mode allows most legacy software to run; it co-exists with 64-bit applications under a 64-bit OS.

The IA-32 system-level architecture includes features to assist in the following operations:

- Memory management
- Protection of software modules
- Multitasking
- Exception and interrupt handling
- Multiprocessing
- Cache management
- Hardware resource and power management
- Debugging and performance monitoring

This chapter provides a description of each part of this architecture. It also describes the system registers that are used to set up and control the processor at the system level and gives a brief overview of the processor's system-level (operating system) instructions.

Many features of the system-level architecture are used only by system programmers. However, application programmers may need to read this chapter and the following chapters in order to create a reliable and secure environment for application programs.

This overview and most subsequent chapters of this book focus on protected-mode operation of the IA-32 architecture. IA-32e mode operation of the Intel 64 architecture, as it differs from protected mode operation, is also described.

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset (see Chapter 9, "Processor Management and Initialization"). Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

## 2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE

System-level architecture consists of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt and exception handling, task management, and control of multiple processors.

Figure 2-1 provides a summary of system registers and data structures that applies to 32-bit modes. System registers and data structures that apply to IA-32e mode are shown in Figure 2-2.

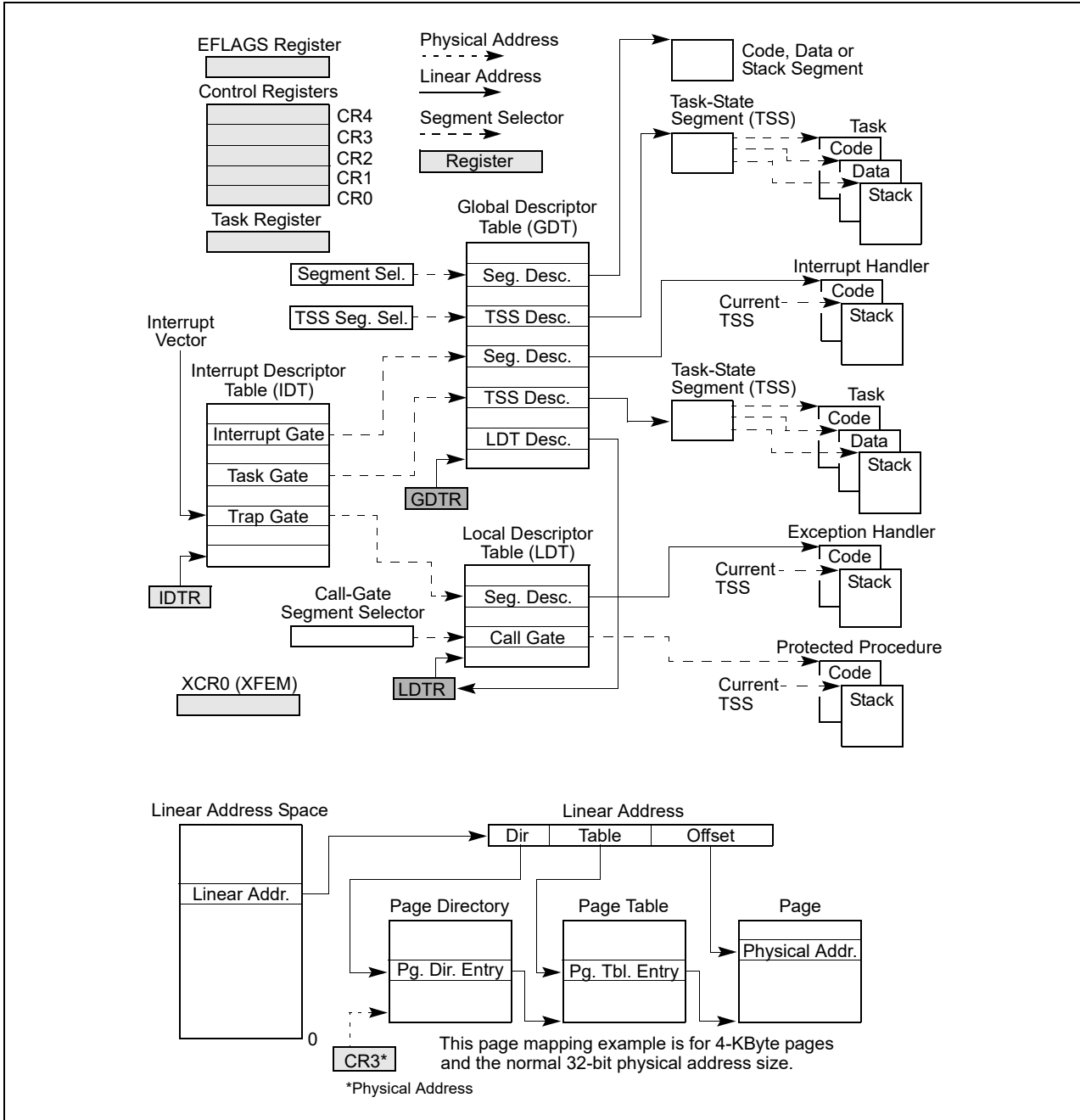


Figure 2-1. IA-32 System-Level Registers and Data Structures

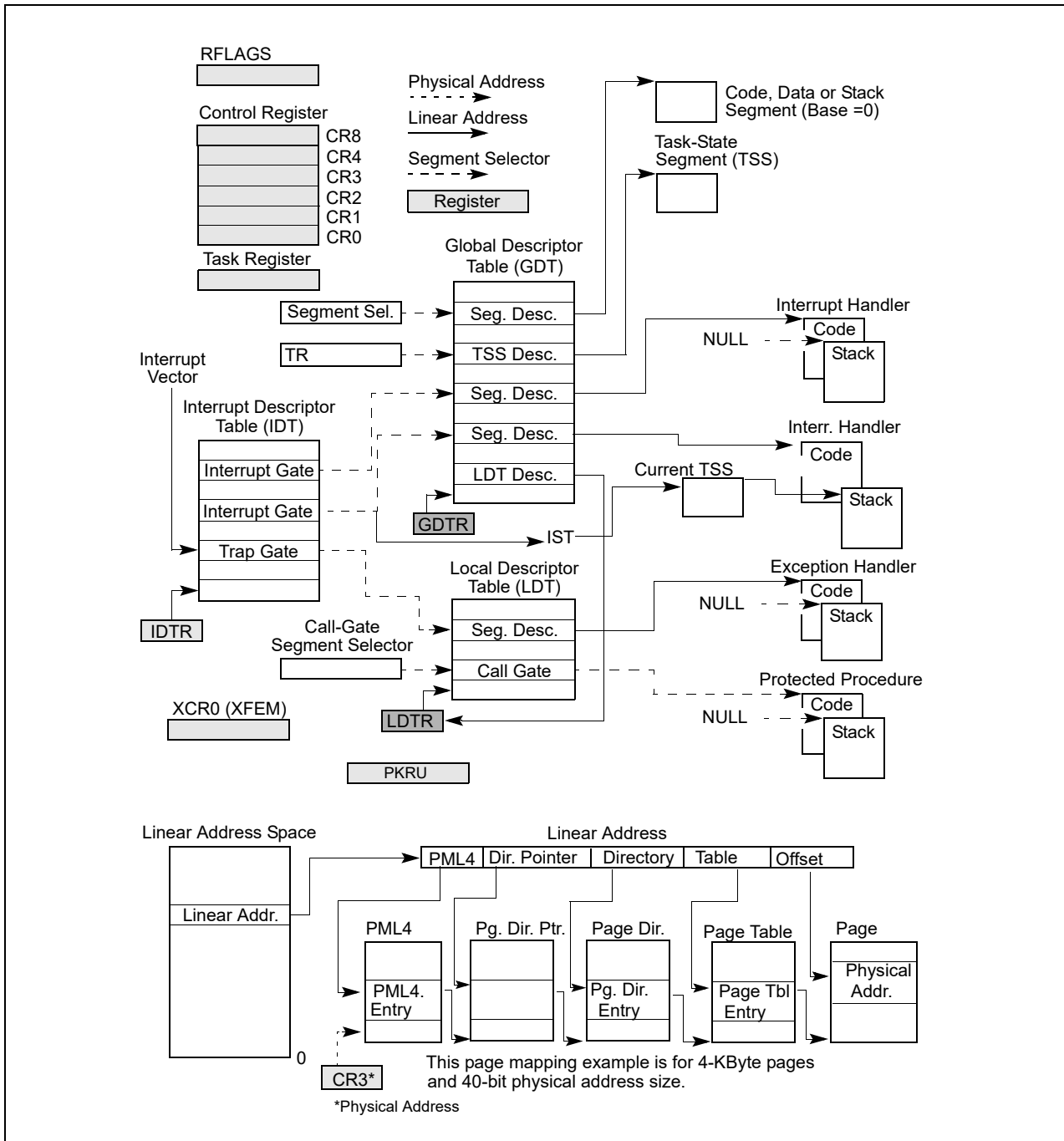


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

### 2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

See Figure 2-1. The solid arrows in the figure indicate a linear address, dashed lines indicate a segment selector, and the dotted arrows indicate a physical address. For simplicity, many of the segment selectors are shown as direct pointers to a segment. However, the actual path from a segment selector to its associated segment is always through a GDT or LDT.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

### 2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode

GDTR and LDTR registers are expanded to 64-bits wide in both IA-32e sub-modes (64-bit mode and compatibility mode). For more information: see Section 3.5.2, "Segment Descriptor Tables in IA-32e Mode."

Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses, (16-byte LDT descriptors hold a 64-bit base address and various attributes). In compatibility mode, descriptors are not expanded.

## 2.1.2 System Segments, Segment Descriptors, and Gates

Besides code, data, and stack segments that make up the execution environment of a program or procedure, the architecture defines two system segments: the task-state segment (TSS) and the LDT. The GDT is not considered a segment because it is not accessed by means of a segment selector and segment descriptor. TSSs and LDTs have segment descriptors defined for them.

The architecture also defines a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures. For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level (more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure<sup>1</sup> supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate.

If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

### 2.1.2.1 Gates in IA-32e Mode

In IA-32e mode, the following descriptors are 16-byte descriptors (expanded to allow a 64-bit base): LDT descriptors, 64-bit TSSs, call gates, interrupt gates, and trap gates.

Call gates facilitate transitions between 64-bit mode and compatibility mode. Task gates are not supported in IA-32e mode. On privilege level changes, stack segment selectors are not read from the TSS. Instead, they are set to NULL.

---

1. The word "procedure" is commonly used in this document as a general term for a logical unit or block of code (such as a program, procedure, function, or routine).

### 2.1.3 Task-State Segments and Task Gates

The TSS (see Figure 2-1) defines the state of the execution environment for a task. It includes the state of general-purpose registers, segment registers, the EFLAGS register, the EIP register, and segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging-structure hierarchy.

All program execution in protected mode happens within the context of a task (called the current task). The segment selector for the TSS for the current task is stored in the task register. The simplest method for switching to a task is to make a call or jump to the new task. Here, the segment selector for the TSS of the new task is given in the CALL or JMP instruction. In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.
2. Loads the task register with the segment selector for the new task.
3. Accesses the new TSS through a segment descriptor in the GDT.
4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.
5. Begins execution of the new task.

A task can also be accessed through a task gate. A task gate is similar to a call gate, except that it provides access (through a segment selector) to a TSS rather than a code segment.

#### 2.1.3.1 Task-State Segments in IA-32e Mode

Hardware task switches are not supported in IA-32e mode. However, TSSs continue to exist. The base address of a TSS is specified by its descriptor.

A 64-bit TSS holds the following information that is important to 64-bit operation:

- Stack pointer addresses for each privilege level
- Pointer addresses for the interrupt stack table
- Offset address of the IO-permission bitmap (from the TSS base)

The task register is expanded to hold 64-bit base addresses in IA-32e mode. See also: Section 7.7, “Task Management in 64-bit Mode.”

### 2.1.4 Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software by means of an INT  $n$ , INTO, INT3, INT1, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

#### 2.1.4.1 Interrupt and Exception Handling IA-32e Mode

In IA-32e mode, interrupt gate descriptors are expanded to 16 bytes to support 64-bit base addresses. This is true for 64-bit mode and compatibility mode.

The IDTR register is expanded to hold a 64-bit base address. Task gates are not supported.



## 2.1.5 Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory.

The location of pages (sometimes called page frames) in physical memory is contained in the paging structures. These structures reside in physical memory (see Figure 2-1 for the case of 32-bit paging).

The base physical address of the paging-structure hierarchy is contained in control register CR3. The entries in the paging structures determine the physical address of the base of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into parts. The parts provide separate offsets into the paging structures and the page frame. A system can have a single hierarchy of paging structures or several. For example, each task can have its own hierarchy.

### 2.1.5.1 Memory Management in IA-32e Mode

In IA-32e mode, physical memory pages are managed by a set of system data structures. In compatibility mode and 64-bit mode, four levels of system data structures are used. These include:

- **The page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. The base physical address of the PML4 is stored in CR3.
- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.
- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.
- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

## 2.1.6 System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights. See also: Section 2.3, "System Flags and Fields in the EFLAGS Register."
- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive. See also: Section 2.5, "Control Registers" and Section 2.6, "Extended Control Registers (Including XCR0)."
- The debug registers (not shown in Figure 2-1) allow the setting of breakpoints for use in debugging programs and systems software. See also: Chapter 17, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."
- The GDTR, LDTR, and IDTR registers contain the linear addresses and sizes (limits) of their respective tables. See also: Section 2.4, "Memory-Management Registers."
- The task register contains the linear address and size of the TSS for the current task. See also: Section 2.4, "Memory-Management Registers."
- Model-specific registers (not shown in Figure 2-1).

The model-specific registers (MSRs) are a group of registers available primarily to operating-system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type ranges (MTRRs).

The number and function of these registers varies among different members of the Intel 64 and IA-32 processor families. See also: Section 9.4, “Model-Specific Registers (MSRs),” and Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

Most systems restrict access to system registers (other than the EFLAGS register) by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

### 2.1.6.1 System Registers in IA-32e Mode

In IA-32e mode, the four system-descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded in hardware to hold 64-bit base addresses. EFLAGS becomes the 64-bit RFLAGS register. CR0–CR4 are expanded to 64 bits. CR8 becomes available. CR8 provides read-write access to the task priority register (TPR) so that the operating system can control the priority classes of external interrupts.

In 64-bit mode, debug registers DR0–DR7 are 64 bits. In compatibility mode, address-matching in DR0–DR3 is also done at 64-bit granularity.

On systems that support IA-32e mode, the extended feature enable register (IA32\_EFER) is available. This model-specific register controls activation of IA-32e mode and other IA-32e mode operations. In addition, there are several model-specific registers that govern IA-32e mode instructions:

- IA32\_KERNEL\_GS\_BASE — Used by SWAPGS instruction.
- IA32\_LSTAR — Used by SYSCALL instruction.
- IA32\_FMASK — Used by SYSCALL instruction.
- IA32\_STAR — Used by SYSCALL and SYSRET instruction.

### 2.1.7 Other System Resources

Besides the system registers and data structures described in the previous sections, system architecture provides the following additional resources:

- Operating system instructions (see also: Section 2.8, “System Instruction Summary”).
- Performance-monitoring counters (not shown in Figure 2-1).
- Internal caches and buffers (not shown in Figure 2-1).

Performance-monitoring counters are event counters that can be programmed to count processor events such as the number of instructions decoded, the number of interrupts received, or the number of cache loads. See also: Chapter 19, “Performance Monitoring Events.”

The processor provides several internal caches and buffers. The caches are used to store both data and instructions. The buffers are used to store things like decoded addresses to system and application segments and write operations waiting to be performed. See also: Chapter 11, “Memory Cache Control.”

## 2.2 MODES OF OPERATION

The IA-32 architecture supports three operating modes and one quasi-operating mode:

- **Protected mode** — This is the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to existing software base.
- **Real-address mode** — This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).
- **System management mode (SMM)** — SMM is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the

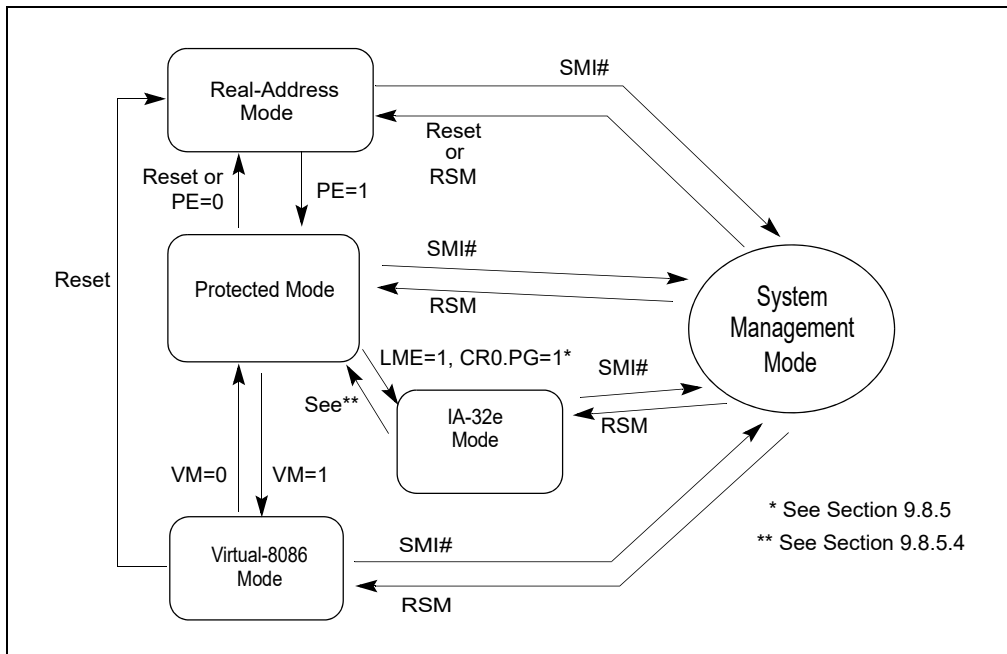
currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

- **Virtual-8086 mode** — In protected mode, the processor supports a quasi-operating mode known as virtual-8086 mode. This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

- **IA-32e mode** — In IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 2-3 shows how the processor moves between operating modes.



**Figure 2-3. Transitions Among the Processor's Operating Modes**

The processor is placed in real-address mode following power-up or a reset. The PE flag in control register CR0 then controls whether the processor is operating in real-address or protected mode. See also: Section 9.9, "Mode Switching." and Section 4.1.2, "Paging-Mode Enabling."

The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. See also: Section 20.2.5, "Entering Virtual-8086 Mode."

The LMA bit (IA32\_EFER.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32\_EFER.LME[bit 8]). See also: Chapter 9, "Processor Management and Initialization."

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

## 2.2.1 Extended Feature Enable Register

The IA32\_EFER MSR provides several fields related to IA-32e mode enabling and operation. It also provides one field that relates to page-access right modification (see Section 4.6, “Access Rights”). The layout of the IA32\_EFER MSR is shown in Figure 2-4.

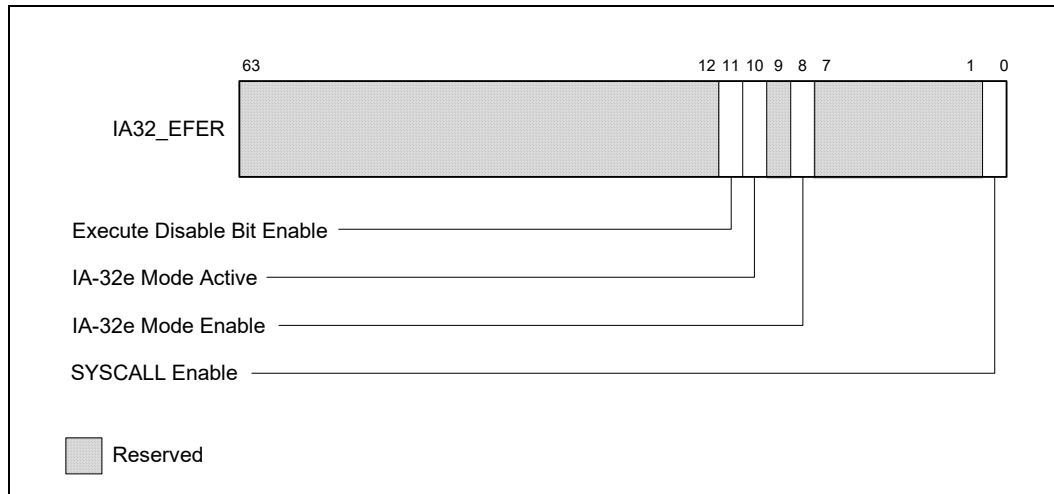


Figure 2-4. IA32\_EFER MSR Layout

Table 2-1. IA32\_EFER MSR Information

Bit	Description
0	<b>SYSCALL Enable: IA32_EFER.SCE (R/W)</b> Enables SYSCALL/SYSRET instructions in 64-bit mode.
7:1	Reserved.
8	<b>IA-32e Mode Enable: IA32_EFER.LME (R/W)</b> Enables IA-32e mode operation.
9	Reserved.
10	<b>IA-32e Mode Active: IA32_EFER.LMA (R)</b> Indicates IA-32e mode is active when set.
11	<b>Execute Disable Bit Enable: IA32_EFER.NXE (R/W)</b> Enables page access restriction by preventing instruction fetches from PAE pages with the XD bit set (See Section 4.6).
63:12	Reserved.

## 2.3 SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER

The system flags and IOPL field of the EFLAGS register control I/O, maskable hardware interrupts, debugging, task switching, and the virtual-8086 mode (see Figure 2-5). Only privileged code (typically operating system or executive code) should be allowed to modify these bits.

The system flags and IOPL are:

**TF**     **Trap (bit 8)** — Set to enable single-step mode for debugging; clear to disable single-step mode. In single-step mode, the processor generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If an application program sets the TF flag using a

POPF, POPFD, or IRET instruction, a debug exception is generated after the instruction that follows the POPF, POPFD, or IRET.

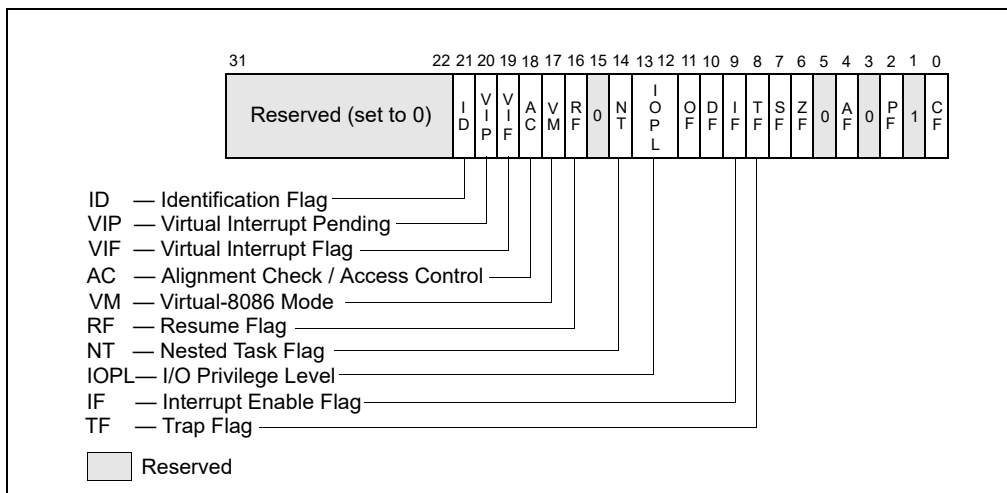


Figure 2-5. System Flags in the EFLAGS Register

**IF** **Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 6.3.2, “Maskable Hardware Interrupts”). The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.

**IOPL** **I/O privilege level field (bits 12 and 13)** — Indicates the I/O privilege level (IOPL) of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

The IOPL is also one of the mechanisms that controls the modification of the IF flag and the handling of interrupts in virtual-8086 mode when virtual mode extensions are in effect (when CR4.VME = 1). See also: Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**NT** **Nested task (bit 14)** — Controls the chaining of interrupted and called tasks. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. It examines and modifies this flag on returns from a task initiated with the IRET instruction. The flag can be explicitly set or cleared with the POPF/POPFD instructions; however, changing to the state of this flag can generate unexpected exceptions in application programs.

See also: Section 7.4, “Task Linking.”

**RF** **Resume (bit 16)** — Controls the processor’s response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints (although other exception conditions can cause an exception to be generated). When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debug software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with IRETD (to prevent the instruction breakpoint from causing another debug exception). The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See also: Section 17.3.1.1, “Instruction-Breakpoint Exception Condition.”

**VM** **Virtual-8086 mode (bit 17)** — Set to enable virtual-8086 mode; clear to return to protected mode.

See also: Section 20.2.1, “Enabling Virtual-8086 Mode.”

- AC Alignment check or access control (bit 18)** — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. An alignment-check exception is generated when reference is made to an unaligned operand, such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. Alignment-check exceptions are generated only in user mode (privilege level 3). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate this exception even when caused by instructions executed in user-mode.
- The alignment-check exception can be used to check alignment of data. This is useful when exchanging data with processors which require all data to be aligned. The alignment-check exception can also be used by interpreters to flag some pointers as special by misaligning the pointer. This eliminates overhead of checking each pointer and only handles the special pointer when used.
- If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1. See Section 4.6, “Access Rights.”
- VIF Virtual Interrupt (bit 19)** — Contains a virtual image of the IF flag. This flag is used in conjunction with the VIP flag. The processor only recognizes the VIF flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. (The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.)
- See also: Section 20.3.3.5, “Method 6: Software Interrupt Handling,” and Section 20.4, “Protected-Mode Virtual Interrupts.”
- VIP Virtual interrupt pending (bit 20)** — Set by software to indicate that an interrupt is pending; cleared to indicate that no interrupt is pending. This flag is used in conjunction with the VIF flag. The processor reads this flag but never modifies it. The processor only recognizes the VIP flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.
- See Section 20.3.3.5, “Method 6: Software Interrupt Handling,” and Section 20.4, “Protected-Mode Virtual Interrupts.”
- ID Identification (bit 21)** — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

### 2.3.1 System Flags and Fields in IA-32e Mode

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. System flags in RFLAGS (64-bit mode) or EFLAGS (compatibility mode) are shown in Figure 2-5.

In IA-32e mode, the processor does not allow the VM bit to be set because virtual-8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit. The processor does, however, allow software to set the NT bit (note that an IRET causes a general protection fault in IA-32e mode if the NT bit is set).

In IA-32e mode, the SYSCALL/SYSCALL instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save/restore EFLAGS/RFLAGS.

## 2.4 MEMORY-MANAGEMENT REGISTERS

The processor provides four memory-management registers (GDTR, LDTR, IDTR, and TR) that specify the locations of the data structures which control segmented memory management (see Figure 2-6). Special instructions are provided for loading and storing these registers.

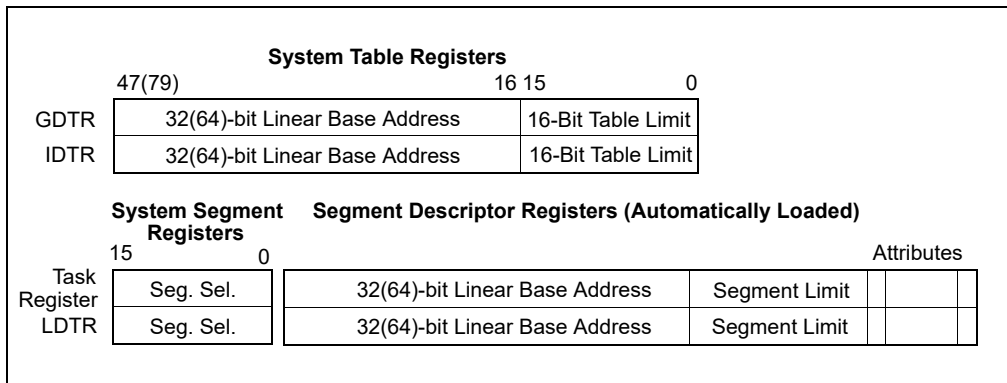


Figure 2-6. Memory Management Registers

### 2.4.1 Global Descriptor Table Register (GDTR)

The GDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and the 16-bit table limit for the GDT. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table.

The LGDT and SGDT instructions load and store the GDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. A new base address must be loaded into the GDTR as part of the processor initialization process for protected-mode operation.

See also: Section 3.5.1, "Segment Descriptor Tables."

### 2.4.2 Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the LDT. The base address specifies the linear address of byte 0 of the LDT segment; the segment limit specifies the number of bytes in the segment. See also: Section 3.5.1, "Segment Descriptor Tables."

The LLDT and SLDT instructions load and store the segment selector part of the LDTR register, respectively. The segment that contains the LDT must have a segment descriptor in the GDT. When the LLDT instruction loads a segment selector in the LDTR: the base address, limit, and descriptor attributes from the LDT descriptor are automatically loaded in the LDTR.

When a task switch occurs, the LDTR is automatically loaded with the segment selector and descriptor for the LDT for the new task. The contents of the LDTR are not automatically saved prior to writing the new LDT information into the register.

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

### 2.4.3 IDTR Interrupt Descriptor Table Register

The IDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and 16-bit table limit for the IDT. The base address specifies the linear address of byte 0 of the IDT; the table limit specifies the number of bytes in the table. The LIDT and SIDT instructions load and store the IDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. The base address and limit in the register can then be changed as part of the processor initialization process.

See also: Section 6.10, "Interrupt Descriptor Table (IDT)."

## 2.4.4 Task Register (TR)

The task register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the TSS of the current task. The selector references the TSS descriptor in the GDT. The base address specifies the linear address of byte 0 of the TSS; the segment limit specifies the number of bytes in the TSS. See also: Section 7.2.4, “Task Register.”

The LTR and STR instructions load and store the segment selector part of the task register, respectively. When the LTR instruction loads a segment selector in the task register, the base address, limit, and descriptor attributes from the TSS descriptor are automatically loaded into the task register. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH.

When a task switch occurs, the task register is automatically loaded with the segment selector and descriptor for the TSS for the new task. The contents of the task register are not automatically saved prior to writing the new TSS information into the register.

## 2.5 CONTROL REGISTERS

Control registers (CR0, CR1, CR2, CR3, and CR4; see Figure 2-7) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode.

In 64-bit mode, control registers are expanded to 64 bits. The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored. The following is also true:

- The control registers can be read and loaded (or modified) using the move-to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating-system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.
- Bits 63:32 of CR0 and CR4 are reserved and must be written with zeros. Writing a nonzero value to any of the upper 32 bits results in a general-protection exception, #GP(0).
- All 64 bits of CR2 are writable by software.
- Bits 51:40 of CR3 are reserved and must be 0.
- The MOV CRn instructions do not check that addresses written to CR2 and CR3 are within the linear-address or physical-address limitations of the implementation.
- Register CR8 is available in 64-bit mode only.

The control registers are summarized below, and each architecturally defined control field in these control registers is described individually. In Figure 2-7, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0).

- **CR0** — Contains system control flags that control operating mode and states of the processor.
- **CR1** — Reserved.
- **CR2** — Contains the page-fault linear address (the linear address that caused a page fault).
- **CR3** — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor’s internal data caches (they do not control TLB caching of page-directory information).

When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table. In IA-32e mode, the CR3 register contains the base address of the PML4 table.

See also: Chapter 4, “Paging.”

- **CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities.



- **CR8** — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

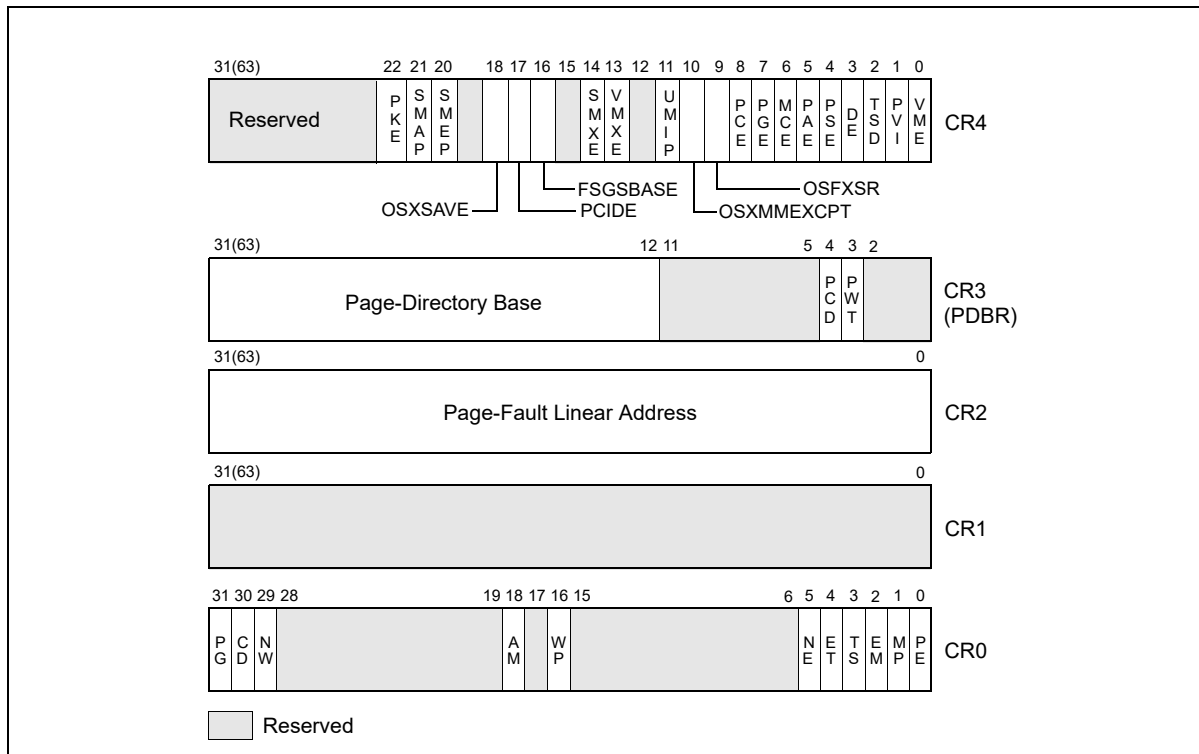


Figure 2-7. Control Registers

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

**CR0.PG**

**Paging (bit 31 of CR0)** — Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, "Paging."

On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

**CR0.CD**

**Cache Disable (bit 30 of CR0)** — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor’s internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur.

See also: Section 11.5.3, "Preventing Caching," and Section 11.5, "Cache Control."

**CR0.NW**

**Not Write-through (bit 29 of CR0)** — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

## CR0.AM

**Alignment Mask (bit 18 of CR0)** — Enables automatic alignment checking when set; disables alignment checking when clear. Alignment checking is performed only when the AM flag is set, the AC flag in the EFLAGS register is set, CPL is 3, and the processor is operating in either protected or virtual-8086 mode.

## CR0.WP

**Write Protect (bit 16 of CR0)** — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.

## CR0.NE

**Numeric Error (bit 5 of CR0)** — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.

The FERR# pin is intended to drive an input to an external interrupt controller (the FERR# pin emulates the ERROR# pin of the Intel 287 and Intel 387 DX math coprocessors). The NE flag, IGNNE# pin, and FERR# pin are used with external logic to implement PC-style error reporting. Using FERR# and IGNNE# to handle floating-point exceptions is deprecated by modern operating systems; this non-native approach also limits newer processors to operate with one logical processor active.

See also: Section 8.7, "Handling x87 FPU Exceptions in Software" in Chapter 8, "Programming with the x87 FPU," and Appendix A, "EFLAGS Cross-Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## CR0.ET

**Extension Type (bit 4 of CR0)** — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

## CR0.TS

**Task Switched (bit 3 of CR0)** — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

- If the TS flag is set and the EM flag (bit 2 of CR0) is clear, a device-not-available exception (#NM) is raised prior to the execution of any x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction; with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. See the paragraph below for the special case of the WAIT/FWAIT instructions.
- If the TS flag is set and the MP flag (bit 1 of CR0) and EM flag are clear, an #NM exception is not raised prior to the execution of an x87 FPU WAIT/FWAIT instruction.
- If the EM flag is set, the setting of the TS flag has no effect on the execution of x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

Table 2-2 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 12-1 and 13-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction.

The processor does not automatically save the context of the x87 FPU, XMM, and MXCSR registers on a task switch. Instead, it sets the TS flag, which causes the processor to raise an #NM exception whenever it encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction in the instruction stream for the new task (with the exception of the instructions listed above).

The fault handler for the #NM exception can then be used to clear the TS flag (with the CLTS instruction) and save the context of the x87 FPU, XMM, and MXCSR registers. If the task never encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction, the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context is never saved.

**Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS**

CRO Flags			x87 FPU Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute.
0	0	1	#NM Exception	Execute.
0	1	0	Execute	Execute.
0	1	1	#NM Exception	#NM exception.
1	0	0	#NM Exception	Execute.
1	0	1	#NM Exception	Execute.
1	1	0	#NM Exception	Execute.
1	1	1	#NM Exception	#NM exception.

**CR0.EM**

**Emulation (bit 2 of CRO)** — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

When the EM flag is set, execution of an x87 FPU instruction generates a device-not-available exception (#NM). This flag must be set when the processor does not have an internal x87 FPU or is not connected to an external math coprocessor. Setting this flag forces all floating-point instructions to be handled by software emulation. Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the EM, MP, and TS flags.

Also, when the EM flag is set, execution of an MMX instruction causes an invalid-opcode exception (#UD) to be generated (see Table 12-1). Thus, if an IA-32 or Intel 64 processor incorporates MMX technology, the EM flag must be set to 0 to enable execution of MMX instructions.

Similarly for SSE/SSE2/SSE3/SSSE3/SSE4 extensions, when the EM flag is set, execution of most SSE/SSE2/SSE3/SSSE3/SSE4 instructions causes an invalid opcode exception (#UD) to be generated (see Table 13-1). If an IA-32 or Intel 64 processor incorporates the SSE/SSE2/SSE3/SSSE3/SSE4 extensions, the EM flag must be set to 0 to enable execution of these extensions. SSE/SSE2/SSE3/SSSE3/SSE4 instructions not affected by the EM flag include: PAUSE, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

**CR0.MP**

**Monitor Coprocessor (bit 1 of CRO)** — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CRO). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag. Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the MP, EM, and TS flags.

**CR0.PE**

**Protection Enable (bit 0 of CRO)** — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.

See also: Section 9.9, "Mode Switching."

**CR3.PCD**

**Page-level Cache Disable (bit 4 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, "Paging and Memory Typing". This bit is not used if paging is disabled, with PAE paging, or with 4-level paging<sup>2</sup> if CR4.PCIDE=1.

2. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

## CR3.PWT

**Page-level Write-Through (bit 3 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, “Paging and Memory Typing”. This bit is not used if paging is disabled, with PAE paging, or with 4-level paging if CR4.PCIDE=1.

## CR4.VME

**Virtual-8086 Mode Extensions (bit 0 of CR4)** — Enables interrupt- and exception-handling extensions in virtual-8086 mode when set; disables the extensions when clear. Use of the virtual mode extensions can improve the performance of virtual-8086 applications by eliminating the overhead of calling the virtual-8086 monitor to handle interrupts and exceptions that occur while executing an 8086 program and, instead, redirecting the interrupts and exceptions back to the 8086 program’s handlers. It also provides hardware support for a virtual interrupt flag (VIF) to improve reliability of running 8086 programs in multi-tasking and multiple-processor environments.

See also: Section 20.3, “Interrupt and Exception Handling in Virtual-8086 Mode.”

## CR4.PVI

**Protected-Mode Virtual Interrupts (bit 1 of CR4)** — Enables hardware support for a virtual interrupt flag (VIF) in protected mode when set; disables the VIF flag in protected mode when clear.

See also: Section 20.4, “Protected-Mode Virtual Interrupts.”

## CR4.TSD

**Time Stamp Disable (bit 2 of CR4)** — Restricts the execution of the RDTSC instruction to procedures running at privilege level 0 when set; allows RDTSC instruction to be executed at any privilege level when clear. This bit also applies to the RDTSCP instruction if supported (if CPUID.80000001H:EDX[27] = 1).

## CR4.DE

**Debugging Extensions (bit 3 of CR4)** — References to debug registers DR4 and DR5 cause an undefined opcode (#UD) exception to be generated when set; when clear, processor aliases references to registers DR4 and DR5 for compatibility with software written to run on earlier IA-32 processors.

See also: Section 17.2.2, “Debug Registers DR4 and DR5.”

## CR4.PSE

**Page Size Extensions (bit 4 of CR4)** — Enables 4-MByte pages with 32-bit paging when set; restricts 32-bit paging to pages of 4 KBytes when clear.

See also: Section 4.3, “32-Bit Paging.”

## CR4.PAE

**Physical Address Extension (bit 5 of CR4)** — When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.

See also: Chapter 4, “Paging.”

## CR4.MCE

**Machine-Check Enable (bit 6 of CR4)** — Enables the machine-check exception when set; disables the machine-check exception when clear.

See also: Chapter 15, “Machine-Check Architecture.”

## CR4.PGE

**Page Global Enable (bit 7 of CR4)** — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory or page-table entry). Global pages are not flushed from the translation-lookaside buffer (TLB) on a task switch or a write to register CR3.

When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.

See also: Section 4.10, “Caching Translation Information.”

## CR4.PCE

**Performance-Monitoring Counter Enable (bit 8 of CR4)** — Enables execution of the RDPMC instruction for programs or procedures running at any protection level when set; RDPMC instruction can be executed only at protection level 0 when clear.

## CR4.OSFXSR

**Operating System Support for FXSAVE and FXRSTOR instructions (bit 9 of CR4)** — When set, this flag: (1) indicates to software that the operating system supports the use of the FXSAVE and FXRSTOR instructions, (2) enables the FXSAVE and FXRSTOR instructions to save and restore the contents of the XMM and MXCSR registers along with the contents of the x87 FPU and MMX registers, and (3) enables the processor to execute SSE/SSE2/SSE3/SSSE3/SSE4 instructions, with the exception of the PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX registers, but they may not save and restore the contents of the XMM and MXCSR registers. Also, the processor will generate an invalid opcode exception (#UD) if it attempts to execute any SSE/SSE2/SSE3 instruction, with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. The operating system or executive must explicitly set this flag.

## NOTE

CPUID feature flag FXSR indicates availability of the FXSAVE/FXRSTOR instructions. The OSFXSR bit provides operating system software with a means of enabling FXSAVE/FXRSTOR to save/restore the contents of the X87 FPU, XMM and MXCSR registers. Consequently OSFXSR bit indicates that the operating system provides context switch support for SSE/SSE2/SSE3/SSSE3/SSE4.

## CR4.OSXMMEXCPT

**Operating System Support for Unmasked SIMD Floating-Point Exceptions (bit 10 of CR4)** — When set, indicates that the operating system supports the handling of unmasked SIMD floating-point exceptions through an exception handler that is invoked when a SIMD floating-point exception (#XM) is generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3/SSE4.1 SIMD floating-point instructions.

The operating system or executive must explicitly set this flag. If this flag is not set, the processor will generate an invalid opcode exception (#UD) whenever it detects an unmasked SIMD floating-point exception.

## CR4.UMIP

**User-Mode Instruction Prevention (bit 11 of CR4)** — When set, the following instructions cannot be executed if CPL > 0: SGDT, SIDT, SLDT, SMSW, and STR. An attempt at such execution causes a general-protection exception (#GP).

## CR4.VMXE

**VMX-Enable Bit (bit 13 of CR4)** — Enables VMX operation when set. See Chapter 23, "Introduction to Virtual Machine Extensions."

## CR4.SMXE

**SMX-Enable Bit (bit 14 of CR4)** — Enables SMX operation when set. See Chapter 6, "Safer Mode Extensions Reference" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*.

## CR4.FSGSBASE

**FSGSBASE-Enable Bit (bit 16 of CR4)** — Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.

## CR4.PCID

**PCID-Enable Bit (bit 17 of CR4)** — Enables process-context identifiers (PCIDs) when set. See Section 4.10.1, "Process-Context Identifiers (PCIDs)". Can be set only in IA-32e mode (if IA32\_EFER.LMA = 1).

**CR4.OSXSAVE**

**XSAVE and Processor Extended States-Enable Bit (bit 18 of CR4)** — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the operating system supports the use of the XGETBV, XSAVE and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 13, “System Programming for Instruction Set Extensions and Processor Extended States”.

**CR4.SMEP**

**SMEP-Enable Bit (bit 20 of CR4)** — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, “Access Rights”.

**CR4.SMAP**

**SMAP-Enable Bit (bit 21 of CR4)** — Enables supervisor-mode access prevention (SMAP) when set. See Section 4.6, “Access Rights”.

**CR4.PKE**

**Protection-Key-Enable Bit (bit 22 of CR4)** — Enables 4-level paging to associate each linear address with a protection key. The PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written. This bit also enables access to the PKRU register using the RDPKRU and WRPKRU instructions.

**CR8.TPL**

**Task Priority Level (bit 3:0 of CR8)** — This sets the threshold value corresponding to the highest-priority interrupt to be blocked. A value of 0 means all interrupts are enabled. This field is available in 64-bit mode. A value of 15 means all interrupts will be disabled.

## 2.5.1 CPUID Qualification of Control Register Flags

Not all flags in control register CR4 are implemented on all processors. With the exception of the PCE flag, they can be qualified with the CPUID instruction to determine if they are implemented on the processor before they are used.

The CR8 register is available on processors that support Intel 64 architecture.

## 2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0)

If CPUID.01H:ECX.XSAVE[bit 26] is 1, the processor supports one or more **extended control registers (XCRs)**. Currently, the only such register defined is XCR0. This register specifies the set of processor state components for which the operating system provides context management, e.g. x87 FPU state, SSE state, AVX state. The OS programs XCR0 to reflect the features for which it provides context management.

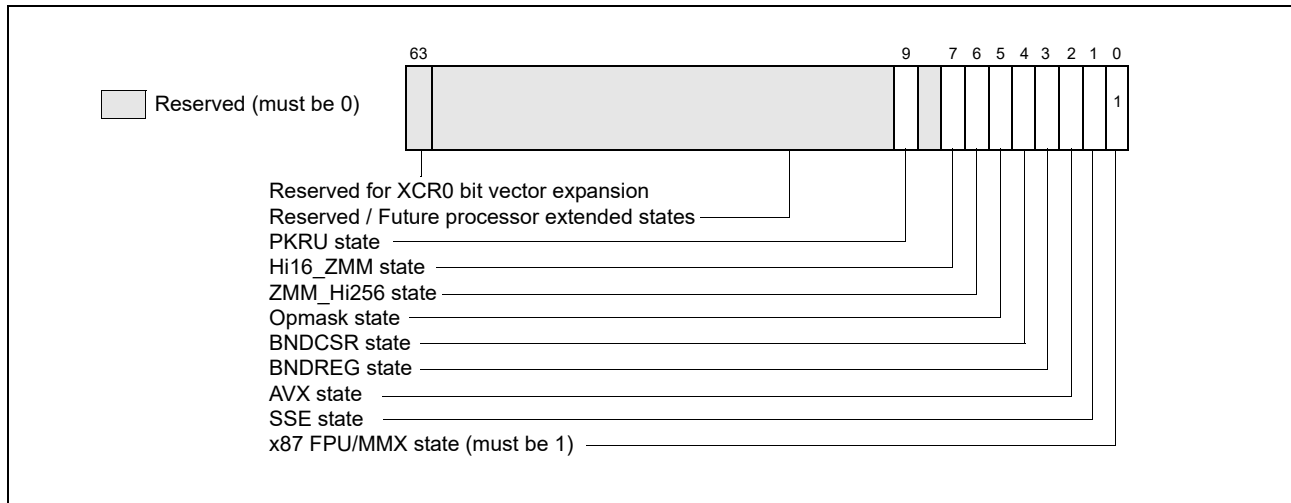


Figure 2-8. XCR0

Software can access XCR0 only if CR4.OSXSAVE[bit 18] = 1. (This bit is also readable as CPUID.01H:ECX.OSXSAVE[bit 27].) Software can use CPUID leaf function 0DH to enumerate the bits in XCR0 that the processor supports (see CPUID instruction in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Each supported state component is represented by a bit in XCR0. System software enables state components by loading an appropriate bit mask value into XCR0 using the XSETBV instruction.

As each bit in XCR0 (except bit 63) corresponds to a processor state component, XCR0 thus provides support for up to 63 sets of processor state components. Bit 63 of XCR0 is reserved for future expansion and will not represent a processor state component.

Currently, XCR0 defines support for the following state components:

- XCR0.X87 (bit 0): This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
- XCR0.SSE (bit 1): If 1, the XSAVE feature set can be used to manage MXCSR and the XMM registers (XMM0-XMM15 in 64-bit mode; otherwise XMM0-XMM7).
- XCR0.AVX (bit 2): If 1, AVX instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the YMM registers (YMM0-YMM15 in 64-bit mode; otherwise YMM0-YMM7).
- XCR0.BNDREG (bit 3): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the bounds registers BND0-BND3.
- XCR0.BNDCSR (bit 4): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the BNDCFGU and BNDSTATUS registers.
- XCR0.opmask (bit 5): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the opmask registers k0-k7.
- XCR0.ZMM\_Hi256 (bit 6): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the lower ZMM registers (ZMM0-ZMM15 in 64-bit mode; otherwise ZMM0-ZMM7).
- XCR0.Hi16\_ZMM (bit 7): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper ZMM registers (ZMM16-ZMM31, only in 64-bit mode).
- XCR0.PKRU (bit 9): If 1, the XSAVE feature set can be used to manage the PKRU register (see Section 2.7).

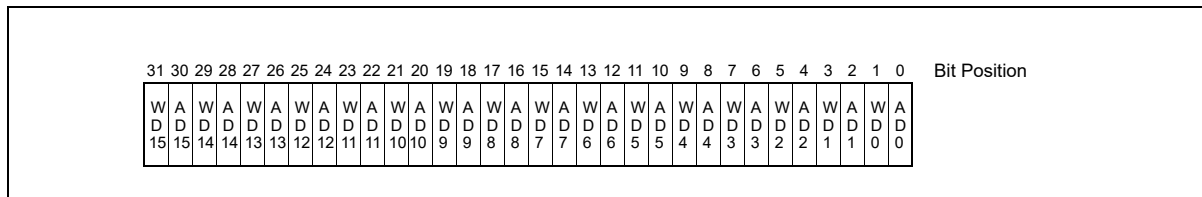
An attempt to use XSETBV to write to XCR0 results in general-protection exceptions (#GP) if it would do any of the following:

- Set a bit reserved in XCR0 for a given processor (as determined by the contents of EAX and EDX after executing CPUID with EAX=0DH, ECX= 0H).
- Clear XCR0.x87.
- Clear XCR0.SSE and set XCR0.AVX.
- Clear XCR0.AVX and set any of XCR0.opmask, XCR0.ZMM\_Hi256, and XCR0.Hi16\_ZMM.
- Set either XCR0.BNDREG and XCR0.BNDCSR while not setting the other.
- Set any of XCR0.opmask, XCR0.ZMM\_Hi256, and XCR0.Hi16\_ZMM while not setting all of them.

After reset, all bits (except bit 0) in XCR0 are cleared to zero; XCR0[0] is set to 1.

## 2.7 PROTECTION KEY RIGHTS REGISTER (PKRU)

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for 4-level paging. The feature allows selective protection of user-mode pages depending on the 4-bit protection key assigned to each page. The **protection key rights register for user pages (PKRU)** allows software to specify the access rights for each protection key.



**Figure 2-9. Protection Key Rights Register for User Pages (PKRU)**

The layout of the PKRU register is shown in Figure 2-9. It contains 16 pairs of disable controls to prevent data accesses to user-mode linear addresses based on their protection keys. Each protection key  $i$  is associated with two bits in the PKRU register:

- Bit  $2i$ , shown as “AD” (access disable): if set, the processor prevents any data accesses to user-mode linear addresses with protection key  $i$ .
- Bit  $2i+1$ , shown as “WD” (write disable): if set, the processor prevents write accesses to user-mode linear addresses with protection key  $i$ .

See Section 4.6.2, “Protection Keys,” for details of how the processor uses the PKRU register to control accesses to user-mode linear addresses.

## 2.8 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-3 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.



**Table 2-3. Summary of System Instructions**

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	If CR4.UMIP = 1
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	If CR4.UMIP = 1
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	If CR4.UMIP = 1
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	If CR4.UMIP = 1
MOV CR <sub>n</sub>	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	If CR4.UMIP = 1
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CRO	No	Yes
ARPL	Adjust RPL	Yes <sup>1,5</sup>	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR <sub>n</sub>	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR <sup>3</sup>	Read Model-Specific Registers	No	Yes
WRMSR <sup>3</sup>	Write Model-Specific Registers	No	Yes
RDPMC <sup>4</sup>	Read Performance-Monitoring Counter	Yes	Yes <sup>2</sup>
RDTSC <sup>3</sup>	Read Time-Stamp Counter	Yes	Yes <sup>2</sup>
RDTSCP <sup>7</sup>	Read Serialized Time-Stamp Counter	Yes	Yes <sup>2</sup>

Table 2-3. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?
XGETBV	Return the state of XCRO	Yes	No
XSETBV	Enable one or more processor extended states	No <sup>6</sup>	Yes

**NOTES:**

1. Useful to application programs running at a CPL of 1 or 2.
2. The TSD and PCE flags in control register CR4 control access to these instructions by application programs running at a CPL of 3.
3. These instructions were introduced into the IA-32 Architecture with the Pentium processor.
4. This instruction was introduced into the IA-32 Architecture with the Pentium Pro processor and the Pentium processor with MMX technology.
5. This instruction is not supported in 64-bit mode.
6. Application uses XGETBV to query which set of processor extended states are enabled.
7. RDTSCP is introduced in Intel Core i7 processor.

## 2.8.1 Loading and Storing System Registers

The GDTR, LDTR, IDTR, and TR registers each have a load and store instruction for loading data into and storing data from the register:

- **LGDT (Load GDTR Register)** — Loads the GDT base address and limit from memory into the GDTR register.
- **SGDT (Store GDTR Register)** — Stores the GDT base address and limit from the GDTR register into memory.
- **LIDT (Load IDTR Register)** — Loads the IDT base address and limit from memory into the IDTR register.
- **SIDT (Store IDTR Register)** — Stores the IDT base address and limit from the IDTR register into memory.
- **LLDT (Load LDTR Register)** — Loads the LDT segment selector and segment descriptor from memory into the LDTR. (The segment selector operand can also be located in a general-purpose register.)
- **SLDT (Store LDTR Register)** — Stores the LDT segment selector from the LDTR register into memory or a general-purpose register.
- **LTR (Load Task Register)** — Loads segment selector and segment descriptor for a TSS from memory into the task register. (The segment selector operand can also be located in a general-purpose register.)
- **STR (Store Task Register)** — Stores the segment selector for the current task TSS from the task register into memory or a general-purpose register.

The LMSW (load machine status word) and SMSW (store machine status word) instructions operate on bits 0 through 15 of control register CR0. These instructions are provided for compatibility with the 16-bit Intel 286 processor. Programs written to run on 32-bit IA-32 processors should not use these instructions. Instead, they should access the control register CR0 using the MOV CR instruction.

The CLTS (clear TS flag in CR0) instruction is provided for use in handling a device-not-available exception (#NM) that occurs when the processor attempts to execute a floating-point instruction when the TS flag is set. This instruction allows the TS flag to be cleared after the x87 FPU context has been saved, preventing further #NM exceptions. See Section 2.5, “Control Registers,” for more information on the TS flag.

The control registers (CR0, CR1, CR2, CR3, CR4, and CR8) are loaded using the MOV instruction. The instruction loads a control register from a general-purpose register or stores the content of a control register in a general-purpose register.

## 2.8.2 Verifying of Access Privileges

The processor provides several instructions for examining segment selectors and segment descriptors to determine if access to their associated segments is allowed. These instructions duplicate some of the automatic access rights and type checking done by the processor, thus allowing operating-system or executive software to prevent exceptions from being generated.

The ARPL (adjust RPL) instruction adjusts the RPL (requestor privilege level) of a segment selector to match that of the program or procedure that supplied the segment selector. See Section 5.10.4, “Checking Caller Access Privileges (ARPL Instruction)” for a detailed explanation of the function and use of this instruction. Note that ARPL is not supported in 64-bit mode.

The LAR (load access rights) instruction verifies the accessibility of a specified segment and loads access rights information from the segment’s segment descriptor into a general-purpose register. Software can then examine the access rights to determine if the segment type is compatible with its intended use. See Section 5.10.1, “Checking Access Rights (LAR Instruction)” for a detailed explanation of the function and use of this instruction.

The LSL (load segment limit) instruction verifies the accessibility of a specified segment and loads the segment limit from the segment’s segment descriptor into a general-purpose register. Software can then compare the segment limit with an offset into the segment to determine whether the offset lies within the segment. See Section 5.10.3, “Checking That the Pointer Offset Is Within Limits (LSL Instruction)” for a detailed explanation of the function and use of this instruction.

The VERR (verify for reading) and VERW (verify for writing) instructions verify if a selected segment is readable or writable, respectively, at a given CPL. See Section 5.10.2, “Checking Read/Write Rights (VERR and VERW Instructions)” for a detailed explanation of the function and use of these instructions.

### 2.8.3 Loading and Storing Debug Registers

Internal debugging facilities in the processor are controlled by a set of 8 debug registers (DR0-DR7). The MOV instruction allows setup data to be loaded to and stored from these registers.

On processors that support Intel 64 architecture, debug registers DR0-DR7 are 64 bits. In 32-bit modes and compatibility mode, writes to a debug register fill the upper 32 bits with zeros. Reads return the lower 32 bits. In 64-bit mode, the upper 32 bits of DR6-DR7 are reserved and must be written with zeros. Writing one to any of the upper 32 bits causes an exception, #GP(0).

In 64-bit mode, MOV DRn instructions read or write all 64 bits of a debug register (operand-size prefixes are ignored). All 64 bits of DR0-DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0-DR3 are in the limits of the implementation. Address matching is supported only on valid addresses generated by the processor implementation.

### 2.8.4 Invalidating Caches and TLBs

The processor provides several instructions for use in explicitly invalidating its caches and TLB entries. The INVD (invalidate cache with no writeback) instruction invalidates all data and instruction entries in the internal caches and sends a signal to the external caches indicating that they should also be invalidated.

The WBINVD (invalidate cache with writeback) instruction performs the same function as the INVD instruction, except that it writes back modified lines in its internal caches to memory before it invalidates the caches. After invalidating the caches local to the executing logical processor or processor core, WBINVD signals caches higher in the cache hierarchy (caches shared with the invalidating logical processor or core) to write back any data they have in modified state at the time of instruction execution and to invalidate their contents.

Note, non-shared caches may not be written back nor invalidated. In Figure 2-10 below, if code executing on either LP0 or LP1 were to execute a WBINVD, the shared L1 and L2 for LP0/LP1 will be written back and invalidated as will the shared L3. However, the L1 and L2 caches not shared with LP0 and LP1 will not be written back nor invalidated.

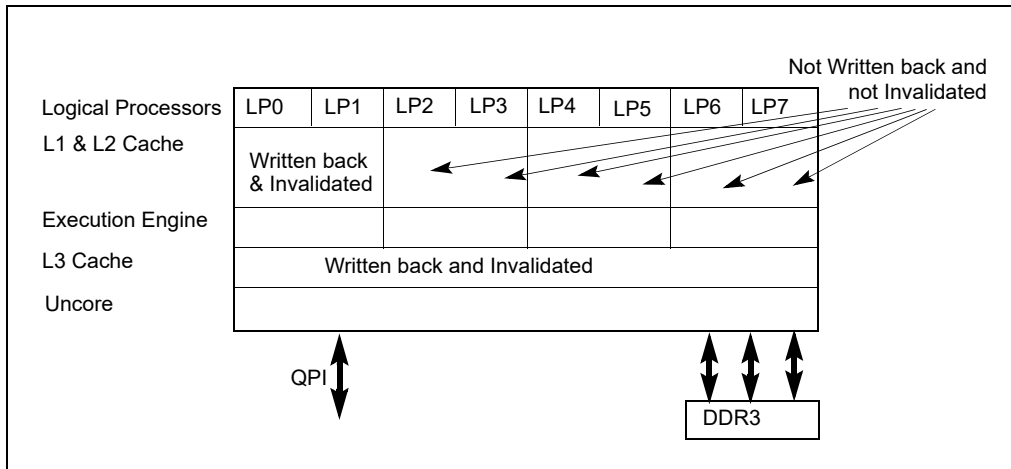


Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy

The INVLPG (invalidate TLB entry) instruction invalidates (flushes) the TLB entry for a specified page.

## 2.8.5 Controlling the Processor

The HLT (halt processor) instruction stops the processor until an enabled interrupt (such as NMI or SMI, which are normally enabled), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal is received. The processor generates a special bus cycle to indicate that the halt mode has been entered.

Hardware may respond to this signal in a number of ways. An indicator light on the front panel may be turned on. An NMI interrupt for recording diagnostic information may be generated. Reset initialization may be invoked (note that the BINIT# pin was introduced with the Pentium Pro processor). If any non-wake events are pending during shutdown, they will be handled after the wake event from shutdown is processed (for example, A20M# interrupts).

The LOCK prefix invokes a locked (atomic) read-modify-write operation when modifying a memory operand. This mechanism is used to allow reliable communications between processors in multiprocessor systems, as described below:

- In the Pentium processor and earlier IA-32 processors, the LOCK prefix causes the processor to assert the LOCK# signal during the instruction. This always causes an explicit bus lock to occur.
- In the Pentium 4, Intel Xeon, and P6 family processors, the locking operation is handled with either a cache lock or bus lock. If a memory access is cacheable and affects only a single cache line, a cache lock is invoked and the system bus and the actual memory location in system memory are not locked during the operation. Here, other Pentium 4, Intel Xeon, or P6 family processors on the bus write-back any modified data and invalidate their caches as necessary to maintain system memory coherency. If the memory access is not cacheable and/or it crosses a cache line boundary, the processor's LOCK# signal is asserted and the processor does not respond to requests for bus control during the locked operation.

The RSM (return from SMM) instruction restores the processor (from a context dump) to the state it was in prior to a system management mode (SMM) interrupt.

## 2.8.6 Reading Performance-Monitoring and Time-Stamp Counters

The RDPMC (read performance-monitoring counter) and RDTSC (read time-stamp counter) instructions allow application programs to read the processor's performance-monitoring and time-stamp counters, respectively. Processors based on Intel NetBurst® microarchitecture have eighteen 40-bit performance-monitoring counters; P6 family processors have two 40-bit counters. Intel® Atom™ processors and most of the processors based on the Intel Core microarchitecture support two types of performance monitoring counters: programmable performance counters similar to those available in the P6 family, and three fixed-function performance monitoring counters.

Details of programmable and fixed-function performance monitoring counters for each processor generation are described in Chapter 18, “Performance Monitoring”.

The programmable performance counters can support counting either the occurrence or duration of events. Events that can be monitored on programmable counters generally are model specific (except for architectural performance events enumerated by CPUID leaf 0AH); they may include the number of instructions decoded, interrupts received, or the number of cache loads. Individual counters can be set up to monitor different events. Use the system instruction WRMSR to set up values in one of the IA32\_PERFEVTSELx MSR, in one of the 45 ESCRs and one of the 18 CCCR MSRs (for Pentium 4 and Intel Xeon processors); or in the PerfEvtSel0 or the PerfEvtSel1 MSR (for the P6 family processors). The RDPMC instruction loads the current count from the selected counter into the EDX:EAX registers.

Fixed-function performance counters record only specific events that are defined in Chapter 19, “Performance Monitoring Events”, and the width/number of fixed-function counters are enumerated by CPUID leaf 0AH.

The time-stamp counter is a model-specific 64-bit counter that is reset to zero each time the processor is reset. If not reset, the counter will increment  $\sim 9.5 \times 10^{16}$  times per year when the processor is operating at a clock rate of 3GHz. At this clock frequency, it would take over 190 years for the counter to wrap around. The RDTSC instruction loads the current count of the time-stamp counter into the EDX:EAX registers.

See Section 18.1, “Performance Monitoring Overview,” and Section 17.17, “Time-Stamp Counter,” for more information about the performance monitoring and time-stamp counters.

The RDTSC instruction was introduced into the IA-32 architecture with the Pentium processor. The RDPMC instruction was introduced into the IA-32 architecture with the Pentium Pro processor and the Pentium processor with MMX technology. Earlier Pentium processors have two performance-monitoring counters, but they can be read only with the RDMSR instruction, and only at privilege level 0.

### 2.8.6.1 Reading Counters in 64-Bit Mode

In 64-bit mode, RDTSC operates the same as in protected mode. The count in the time-stamp counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

RDPMC requires an index to specify the offset of the performance-monitoring counter. In 64-bit mode for Pentium 4 or Intel Xeon processor families, the index is specified in ECX[30:0]. The current count of the performance-monitoring counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

## 2.8.7 Reading and Writing Model-Specific Registers

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions allow a processor’s 64-bit model-specific registers (MSRs) to be read and written, respectively. The MSR to be read or written is specified by the value in the ECX register.

RDMSR reads the value from the specified MSR to the EDX:EAX registers; WRMSR writes the value in the EDX:EAX registers to the specified MSR. RDMSR and WRMSR were introduced into the IA-32 architecture with the Pentium processor.

See Section 9.4, “Model-Specific Registers (MSRs),” for more information.

### 2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode

RDMSR and WRMSR require an index to specify the address of an MSR. In 64-bit mode, the index is 32 bits; it is specified using ECX.

## 2.8.8 Enabling Processor Extended States

The XSETBV instruction is required to enable OS support of individual processor extended states in XCR0 (see Section 2.6).

## 8. Updates to Chapter 3, Volume 3A

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

-----  
Change to this chapter: Update to Section 3.4.3 "Segment Registers".

## CHAPTER 3

# PROTECTED-MODE MEMORY MANAGEMENT

---

This chapter describes the Intel 64 and IA-32 architecture's protected-mode memory management facilities, including the physical memory requirements, segmentation mechanism, and paging mechanism.

See also: Chapter 5, "Protection" (for a description of the processor's protection mechanism) and Chapter 20, "8086 Emulation" (for a description of memory addressing protection in real-address and virtual-8086 modes).

### 3.1 MEMORY MANAGEMENT OVERVIEW

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. **There is no mode bit to disable segmentation.** The use of paging, however, is optional.

These two mechanisms (segmentation and paging) can be configured to support simple single-program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

As shown in Figure 3-1, segmentation provides a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS or LDT). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and insures that one program does not interfere with the execution of another program by writing into the other program's segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

All the segments in a system are contained in the processor's linear address space. To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

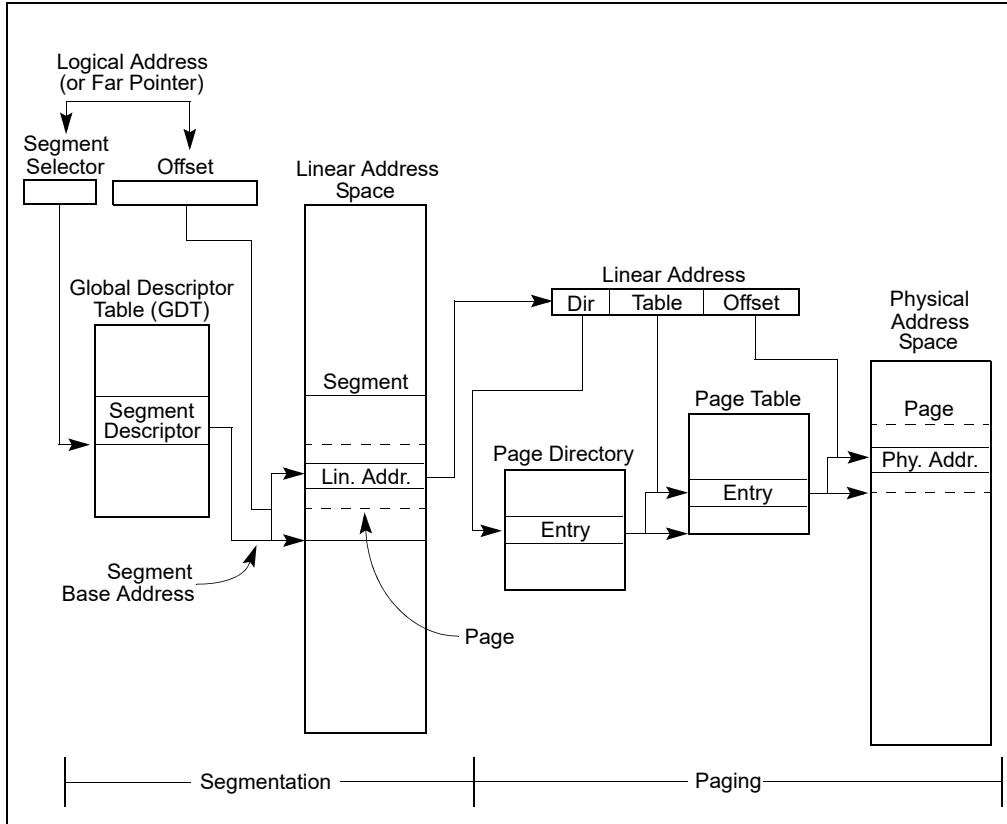


Figure 3-1. Segmentation and Paging

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of “virtualizing” the linear address space is needed. This virtualization of the linear address space is handled through the processor’s paging mechanism.

Paging supports a “virtual memory” environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

When paging is implemented properly in the operating-system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit IA-32 processors can be paged (transparently) when they are run in virtual-8086 mode.

### 3.2 USING SEGMENTS

The segmentation mechanism supported by the IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect



programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

### 3.2.1 Basic Flat Model

The simplest memory model for a system is the basic “flat model,” in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF\_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

### 3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

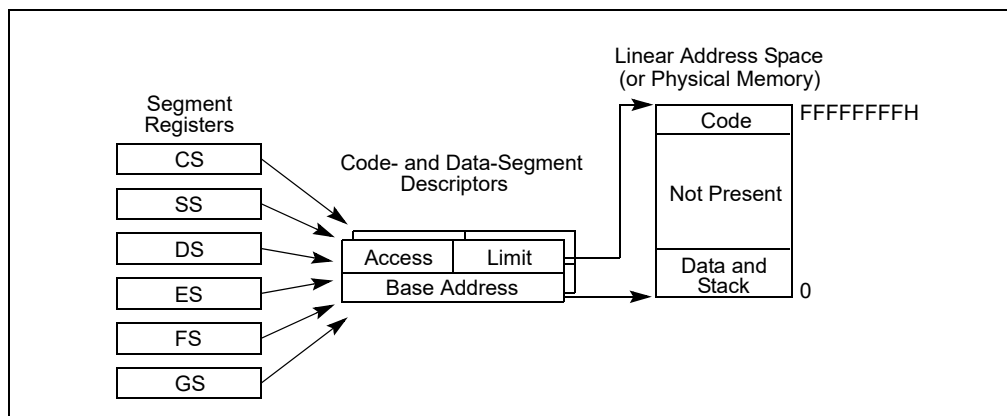


Figure 3-2. Flat Model

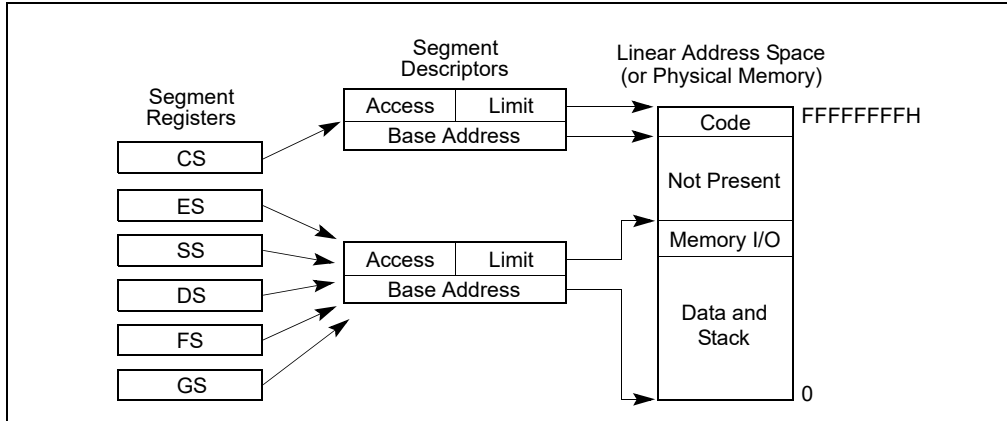


Figure 3-3. Protected Flat Model

More complexity can be added to this protected flat model to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

### 3.2.3 Multi-Segment Model

A multi-segment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

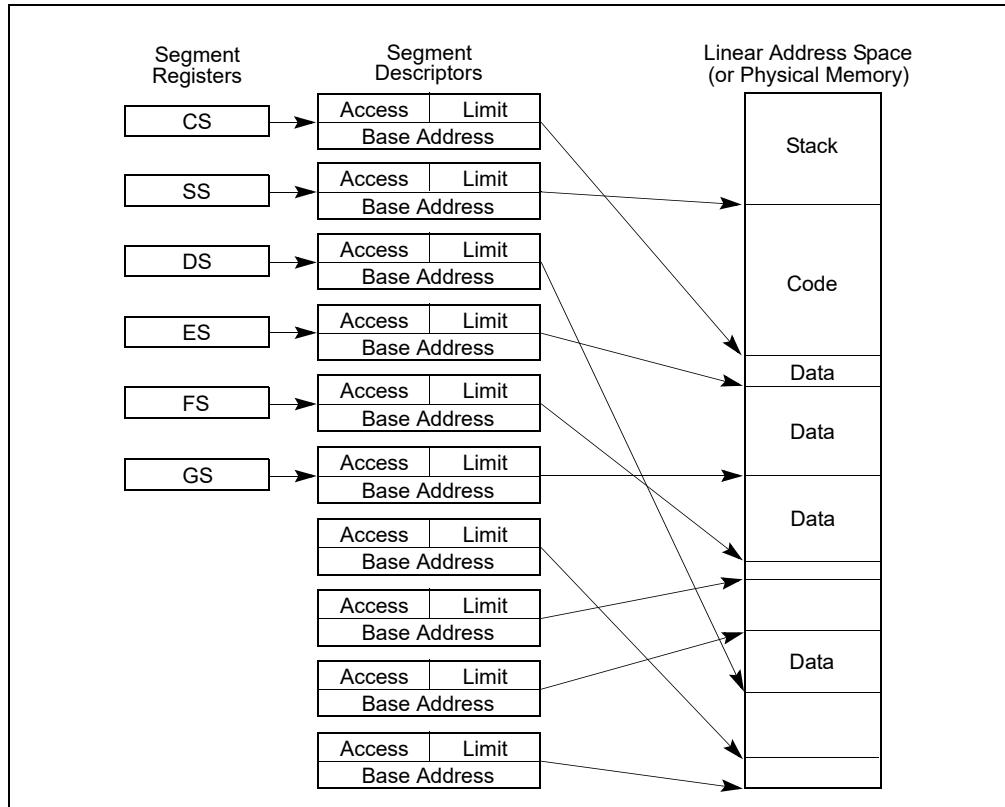


Figure 3-4. Multi-Segment Model

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

### 3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, 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 using legacy 16-bit or 32-bit protected mode semantics.

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 FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

### 3.2.5 Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor's paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-

protection facilities. For example, it lets read-write protection be enforced on a page-by-page basis. The paging mechanism also provides two-level user-supervisor protection that can also be specified on a page-by-page basis.

### 3.3 PHYSICAL ADDRESS SPACE

In protected mode, the IA-32 architecture provides a normal physical address space of 4 GBytes ( $2^{32}$  bytes). This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFFFFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

Starting with the Pentium Pro processor, the IA-32 architecture also supports an extension of the physical address space to  $2^{36}$  bytes (64 GBytes); with a maximum physical address of FFFFFFFFH. This extension is invoked in either of two ways:

- Using the physical address extension (PAE) flag, located in bit 5 of control register CR4.
- Using the 36-bit page size extension (PSE-36) feature (introduced in the Pentium III processors).

Physical address support has since been extended beyond 36 bits. See Chapter 4, "Paging" for more information about 36-bit physical addressing.

#### 3.3.1 Intel® 64 Processors and Physical Address Space

On processors that support Intel 64 architecture (CPUID.8000001H:EDX[29] = 1), the size of the physical address range is implementation-specific and indicated by CPUID.8000008H:EAX[bits 7-0].

For the format of information returned in EAX, see "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. See also: Chapter 4, "Paging."

### 3.4 LOGICAL AND LINEAR ADDRESSES

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging.

Even with the minimum use of segments, every byte in the processor's address space is accessed with a logical address. A logical address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5). The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor's linear address space. Like the physical address space, the linear address space is a flat (unsegmented),  $2^{32}$ -byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. (This step is needed only when a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

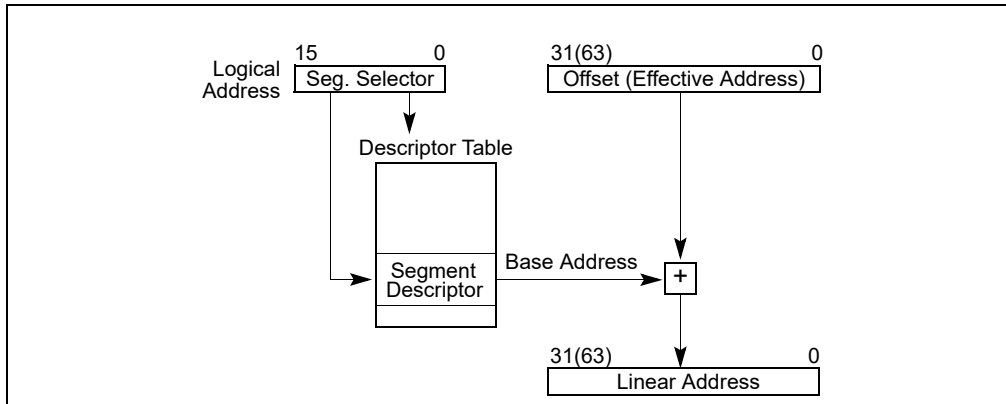


Figure 3-5. Logical Address to Linear Address Translation

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor’s address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, “Paging.”

### 3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

### 3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

**Index** (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

**TI (table indicator) flag** (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

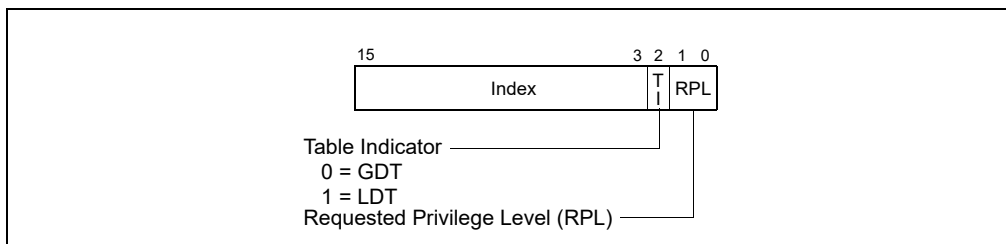


Figure 3-6. Segment Selector

**Requested Privilege Level (RPL)**

(Bits 0 and 1) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level. See Section 5.5, "Privilege Levels", for a description of the relationship of the RPL to the CPL of the executing program (or task) and the descriptor privilege level (DPL) of the descriptor the segment selector points to.

The first entry of the GDT is not used by the processor. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a "null segment selector." The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers. Loading the CS or SS register with a null segment selector causes a general-protection exception (#GP) to be generated.

Segment selectors are visible to application programs as part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs.

**3.4.3 Segment Registers**

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7). Each of these segment registers support a specific kind of memory reference (code, stack, or data). For virtually any kind of program execution to take place, at least the code-segment (CS), data-segment (DS), and stack-segment (SS) registers must be loaded with valid segment selectors. The processor also provides three additional data-segment registers (ES, FS, and GS), which can be used to make additional data segments available to the currently executing program (or task).

For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. So, although a system can define thousands of segments, only 6 can be available for immediate use. Other segments can be made available by loading their segment selectors into these registers during program execution.

Visible Part		Hidden Part	
Segment Selector	Base Address, Limit, Access Information		
			CS
			SS
			DS
			ES
			FS
			GS

**Figure 3-7. Segment Registers**

Every segment register has a "visible" part and a "hidden" part. (The hidden part is sometimes referred to as a "descriptor cache" or a "shadow register.") When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of load instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, POP, LDS, LES, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers.
2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INT *n*, INTO, INT3, and INT1 instructions. These instructions change

the contents of the CS register (and sometimes other segment registers) as an incidental part of their operation.

The MOV instruction can also be used to store the visible part of a segment register in a general-purpose register.

### 3.4.4 Segment Loading Instructions in IA-32e Mode

Because ES, DS, and SS segment registers are not used in 64-bit mode, their fields (base, limit, and attribute) in segment descriptor registers are ignored. Some forms of segment load instructions are also invalid (for example, LDS, POP ES). Address calculations that reference the ES, DS, or SS segments are treated as if the segment base is zero.

The processor checks that all linear-address references are in canonical form instead of performing limit checks. Mode switching does not change the contents of the segment registers or the associated descriptor registers. These registers are also not changed during 64-bit mode execution, unless explicit segment loads are performed.

In order to set up compatibility mode for an application, segment-load instructions (MOV to Sreg, POP Sreg) work normally in 64-bit mode. An entry is read from the system descriptor table (GDT or LDT) and is loaded in the hidden portion of the segment register. The descriptor-register base, limit, and attribute fields are all loaded. However, the contents of the data and stack segment selector and the descriptor registers are ignored.

When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the linear address calculation: (FS or GS).base + index + displacement. FS.base and GS.base are then expanded to the full linear-address size supported by the implementation. The resulting effective address calculation can wrap across positive and negative addresses; the resulting linear address must be canonical.

In 64-bit mode, memory accesses using FS-segment and GS-segment overrides are not checked for a runtime limit nor subjected to attribute-checking. Normal segment loads (MOV to Sreg and POP Sreg) into FS and GS load a standard 32-bit base value in the hidden portion of the segment register. The base address bits above the standard 32 bits are cleared to 0 to allow consistency for implementations that use less than 64 bits.

The hidden descriptor register fields for FS.base and GS.base are physically mapped to MSRs in order to load all address bits supported by a 64-bit implementation. Software with CPL = 0 (privileged software) can load all supported linear-address bits into FS.base or GS.base using WRMSR. Addresses written into the 64-bit FS.base and GS.base registers must be in canonical form. A WRMSR instruction that attempts to write a non-canonical address to those registers causes a #GP fault.

When in compatibility mode, FS and GS overrides operate as defined by 32-bit mode behavior regardless of the value loaded into the upper 32 linear-address bits of the hidden descriptor register base field. Compatibility mode ignores the upper 32 bits when calculating an effective address.

A new 64-bit mode instruction, SWAPGS, can be used to load GS base. SWAPGS exchanges the kernel data structure pointer from the IA32\_KERNEL\_GS\_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access the kernel data structures. An attempt to write a non-canonical value (using WRMSR) to the IA32\_KERNEL\_GS\_BASE MSR causes a #GP fault.

### 3.4.5 Segment Descriptors

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information. Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs. Figure 3-8 illustrates the general descriptor format for all types of segment descriptors.

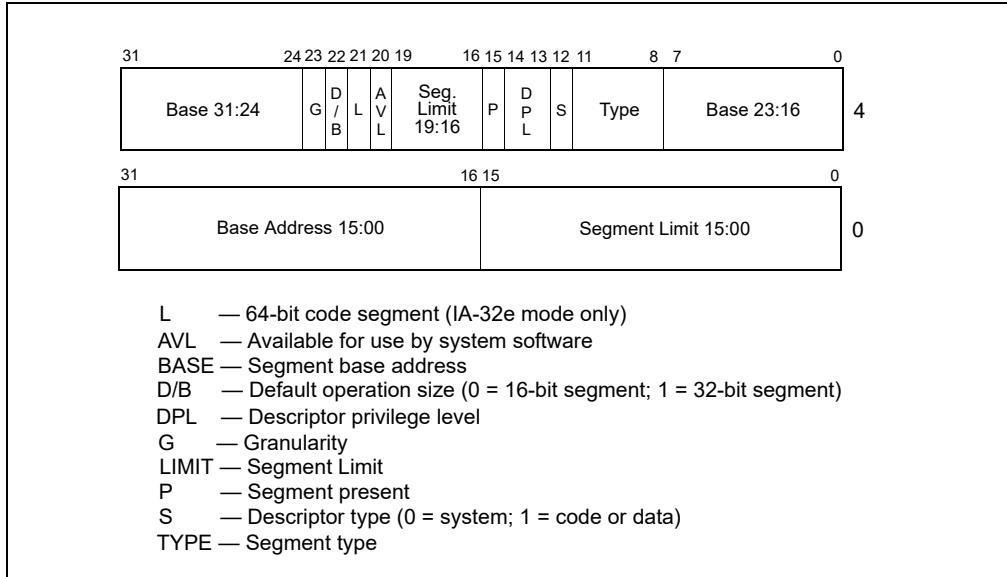


Figure 3-8. Segment Descriptor

The flags and fields in a segment descriptor are as follows:

**Segment limit field**

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segments other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

**Base address fields**

Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

**Type field**

Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.



**S (descriptor type) flag**

Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

**DPL (descriptor privilege level) field**

Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment. See Section 5.5, "Privilege Levels", for a description of the relationship of the DPL to the CPL of the executing code segment and the RPL of a segment selector.

**P (segment-present) flag**

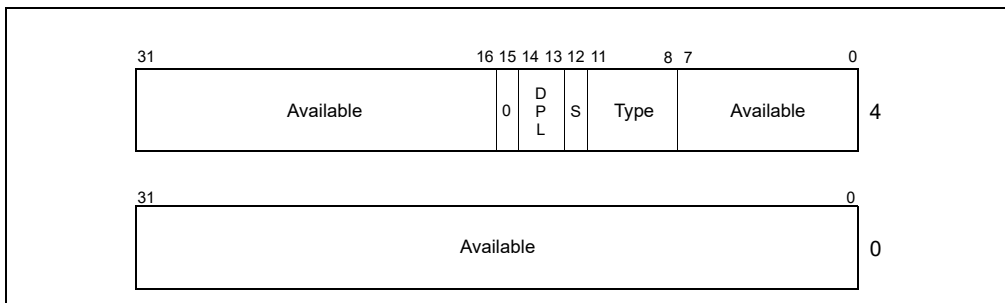
Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

Figure 3-9 shows the format of a segment descriptor when the segment-present flag is clear. When this flag is clear, the operating system or executive is free to use the locations marked "Available" to store its own data, such as information regarding the whereabouts of the missing segment.

**D/B (default operation size/default stack pointer size and/or upper bound) flag**

Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

- **Executable code segment.** The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.
- **Stack segment (data segment pointed to by the SS register).** The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.
- **Expand-down data segment.** The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).



**Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear**

**G (granularity) flag**

Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units. (This flag does not affect the granularity of the base address; it is always byte granular.) When the granularity flag is set, the twelve least significant bits of an offset are not tested when checking the

offset against the segment limit. For example, when the granularity flag is set, a limit of 0 results in valid offsets from 0 to 4095.

**L (64-bit code segment) flag**

In IA-32e mode, bit 21 of the second doubleword of the segment descriptor indicates whether a code segment contains native 64-bit code. A value of 1 indicates instructions in this code segment are executed in 64-bit mode. A value of 0 indicates the instructions in this code segment are executed in compatibility mode. If L-bit is set, then D-bit must be cleared. When not in IA-32e mode or for non-code segments, bit 21 is reserved and should always be set to 0.

**Available and reserved bits**

Bit 20 of the second doubleword of the segment descriptor is available for use by system software.

**3.4.5.1 Code- and Data-Segment Descriptor Types**

When the S (descriptor type) flag in a segment descriptor is set, the descriptor is for either a code or a data segment. The highest order bit of the type field (bit 11 of the second double word of the segment descriptor) then determines whether the descriptor is for a data segment (clear) or a code segment (set).

For data segments, the three low-order bits of the type field (bits 8, 9, and 10) are interpreted as accessed (A), write-enabled (W), and expansion-direction (E). See Table 3-1 for a description of the encoding of the bits in the type field for code and data segments. Data segments can be read-only or read/write segments, depending on the setting of the write-enabled bit.

**Table 3-1. Code- and Data-Segment Types**

Decimal	Type Field				Descriptor Type	Description
	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		<b>C</b>	<b>R</b>	<b>A</b>		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception (#GP). If the size of a stack segment needs to be changed dynamically, the stack segment can be an expand-down data segment (expansion-direction flag set). Here, dynamically changing the segment limit causes stack space to be added to the bottom of

the stack. If the size of a stack segment is intended to remain static, the stack segment may be either an expand-up or expand-down type.

The accessed bit indicates whether the segment has been accessed since the last time the operating-system or executive cleared the bit. The processor sets this bit whenever it loads a segment selector for the segment into a segment register, assuming that the type of memory that contains the segment descriptor supports processor writes. The bit remains set until explicitly cleared. This bit can be used both for virtual memory management and for debugging.

For code segments, the three low-order bits of the type field are interpreted as accessed (A), read enable (R), and conforming (C). Code segments can be execute-only or execute/read, depending on the setting of the read-enable bit. An execute/read segment might be used when constants or other static data have been placed with instruction code in a ROM. Here, data can be read from the code segment either by using an instruction with a CS override prefix or by loading a segment selector for the code segment in a data-segment register (the DS, ES, FS, or GS registers). In protected mode, code segments are not writable.

Code segments can be either conforming or nonconforming. A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general-protection exception (#GP), unless a call gate or task gate is used (see Section 5.8.1, "Direct Calls or Jumps to Code Segments", for more information on conforming and nonconforming code segments). System utilities that do not access protected facilities and handlers for some types of exceptions (such as, divide error or overflow) may be loaded in conforming code segments. Utilities that need to be protected from less privileged programs and procedures should be placed in nonconforming code segments.

### NOTE

Execution cannot be transferred by a call or a jump to a less-privileged (numerically higher privilege level) code segment, regardless of whether the target segment is a conforming or nonconforming code segment. Attempting such an execution transfer will result in a general-protection exception.

All data segments are nonconforming, meaning that they cannot be accessed by less privileged programs or procedures (code executing at numerically higher privilege levels). Unlike code segments, however, data segments can be accessed by more privileged programs or procedures (code executing at numerically lower privilege levels) without using a special access gate.

If the segment descriptors in the GDT or an LDT are placed in ROM, the processor can enter an indefinite loop if software or the processor attempts to update (write to) the ROM-based segment descriptors. To prevent this problem, set the accessed bits for all segment descriptors placed in a ROM. Also, remove operating-system or executive code that attempts to modify segment descriptors located in ROM.

## 3.5 SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.
- Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves "gates," which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS's (task gates).

Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors. Note that system descriptors in IA-32e mode are 16 bytes instead of 8 bytes.

**Table 3-2. System-Segment and Gate-Descriptor Types**

Type Field					Description	
Decimal	11	10	9	8	32-Bit Mode	IA-32e Mode
0	0	0	0	0	Reserved	Reserved
1	0	0	0	1	16-bit TSS (Available)	Reserved
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-bit TSS (Busy)	Reserved
4	0	1	0	0	16-bit Call Gate	Reserved
5	0	1	0	1	Task Gate	Reserved
6	0	1	1	0	16-bit Interrupt Gate	Reserved
7	0	1	1	1	16-bit Trap Gate	Reserved
8	1	0	0	0	Reserved	Reserved
9	1	0	0	1	32-bit TSS (Available)	64-bit TSS (Available)
10	1	0	1	0	Reserved	Reserved
11	1	0	1	1	32-bit TSS (Busy)	64-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate	64-bit Call Gate
13	1	1	0	1	Reserved	Reserved
14	1	1	1	0	32-bit Interrupt Gate	64-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate	64-bit Trap Gate

See also: Section 3.5.1, “Segment Descriptor Tables”, and Section 7.2.2, “TSS Descriptor” (for more information on the system-segment descriptors); see Section 5.8.3, “Call Gates”, Section 6.11, “IDT Descriptors”, and Section 7.2.5, “Task-Gate Descriptor” (for more information on the gate descriptors).

### 3.5.1 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 ( $2^{13}$ ) 8-byte descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

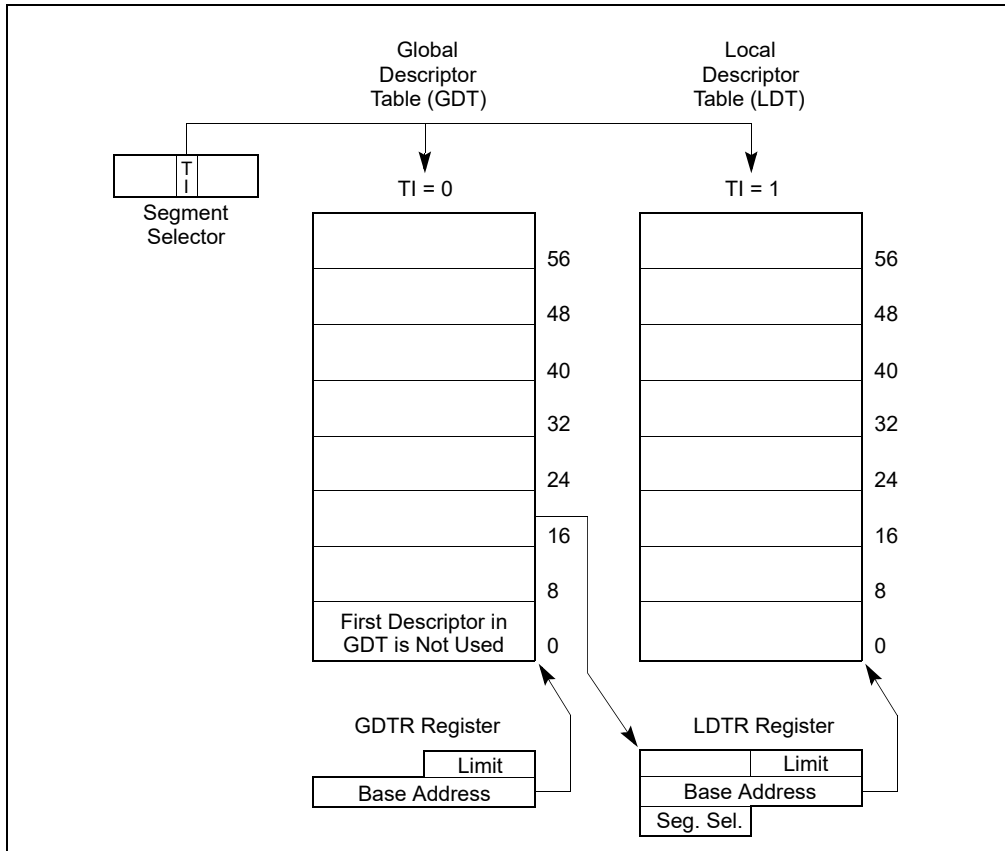


Figure 3-10. Global and Local Descriptor Tables

Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register (see Section 2.4, "Memory-Management Registers"). The base address of the GDT should be aligned on an eight-byte boundary to yield the best processor performance. The limit value for the GDT is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is,  $8N - 1$ ).

The first descriptor in the GDT is not used by the processor. A segment selector to this "null descriptor" does not generate an exception when loaded into a data-segment register (DS, ES, FS, or GS), but it always generates a general-protection exception (#GP) when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

The LDT is located in a system segment of the LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. The segment descriptor for an LDT can be located anywhere in the GDT. See Section 3.5, "System Descriptor Types", for information on the LDT segment-descriptor type.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register (see Section 2.4, "Memory-Management Registers").

When the GDTR register is stored (using the SGDT instruction), a 48-bit "pseudo-descriptor" is stored in memory (see top diagram in Figure 3-11). To avoid alignment check faults in user mode (privilege level 3), the pseudo-descriptor should be located at an odd word address (that is, address MOD 4 is equal to 2). This causes the

processor to store an aligned word, followed by an aligned doubleword. User-mode programs normally do not store pseudo-descriptors, but the possibility of generating an alignment check fault can be avoided by aligning pseudo-descriptors in this way. The same alignment should be used when storing the IDTR register using the SIDT instruction. When storing the LDTR or task register (using the SLDT or STR instruction, respectively), the pseudo-descriptor should be located at a doubleword address (that is, address MOD 4 is equal to 0).

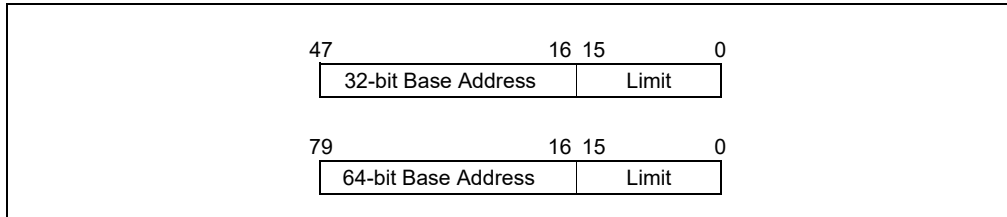


Figure 3-11. Pseudo-Descriptor Formats

### 3.5.2 Segment Descriptor Tables in IA-32e Mode

In IA-32e mode, a segment descriptor table can contain up to 8192 ( $2^{13}$ ) 8-byte descriptors. An entry in the segment descriptor table can be 8 bytes. System descriptors are expanded to 16 bytes (occupying the space of two entries).

GDTR and LDTR registers are expanded to hold 64-bit base address. The corresponding pseudo-descriptor is 80 bits. (see the bottom diagram in Figure 3-11).

The following system descriptors expand to 16 bytes:

- Call gate descriptors (see Section 5.8.3.1, "IA-32e Mode Call Gates")
- IDT gate descriptors (see Section 6.14.1, "64-Bit Mode IDT")
- LDT and TSS descriptors (see Section 7.2.3, "TSS Descriptor in 64-bit mode").

## 9. Updates to Chapter 6, Volume 3A

Change bars show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

---

Changes to this chapter: Update to Table 6-1 "Protected-Mode Exceptions and Interrupts". Updates to Section 6.4.2 "Software-Generated Exceptions", Section 6.8.1 "Masking Maskable Hardware Interrupts", Section 6.8.2 "Masking Instruction Breakpoints", Section 6.8.3 "Masking Exceptions and Interrupts When Switching Stacks", Section 6.12.1.1 "Protection of Exception- and Interrupt-Handler Procedures", Section 6.12.1.2 "Flag Usage By Exception- or Interrupt-Handler Procedure", and Section 6.13 "Error Code". Updates to Interrupt 1—Debug Exception (#DB), Interrupt 3—Breakpoint Exception (#BP), and Interrupt 6—Invalid Opcode Exception (#UD).

# CHAPTER 6

## INTERRUPT AND EXCEPTION HANDLING

---

This chapter describes the interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

Chapter 20, “8086 Emulation,” describes information specific to interrupt and exception mechanisms in real-address and virtual-8086 mode. Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” describes information specific to interrupt and exception mechanisms in IA-32e mode and 64-bit sub-mode.

### 6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the `INT n` instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor’s interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

### 6.2 EXCEPTION AND INTERRUPT VECTORS

To aid in handling exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned a unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT). The table provides the entry point to an exception or interrupt handler (see Section 6.10, “Interrupt Descriptor Table (IDT)”).

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture-defined exceptions and interrupts. Not all of the vector numbers in this range have a currently defined function. The unassigned vector numbers in this range are reserved. Do not use the reserved vector numbers.

Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are not reserved by the Intel 64 and IA-32 architecture. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms (see Section 6.3, “Sources of Interrupts”).



Table 6-1 shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (see Section 6.5, “Exception Classifications”) and indicates whether an error code is saved on the stack for the exception. The source of each predefined exception and the NMI interrupt is also given.

### 6.3 SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

#### 6.3.1 External Interrupts

External interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the local APIC (see Chapter 10, “Advanced Programmable Interrupt Controller (APIC)”). When the local APIC is enabled, the LINT[1:0] pins can be programmed through the APIC’s local vector table (LVT) to be associated with any of the processor’s exception or interrupt vectors.

When the local APIC is global/hardware disabled, these pins are configured as INTR and NMI pins, respectively. Asserting the INTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A (see Section 6.2, “Exception and Interrupt Vectors”). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

**Table 6-1. Protected-Mode Exceptions and Interrupts**

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. <sup>1</sup>
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

**Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)**

15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory. <sup>2</sup>
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent. <sup>3</sup>
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions <sup>4</sup>
20	#VE	Virtualization Exception	Fault	No	EPT violations <sup>5</sup>
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT <i>n</i> instruction.

**NOTES:**

- Processors after the Intel386 processor do not generate this exception.
- This exception was introduced in the Intel486 processor.
- This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
- This exception was introduced in the Pentium III processor.
- This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.

The processor’s local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC’s pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel® Atom™, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 34, “System Management Mode.”

### 6.3.2 Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include all IA-32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local APIC include interrupt vectors 16 through 255.

The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group (see Section 6.8.1, “Masking Maskable Hardware Interrupts”). Note that when interrupts 0 through 15 are delivered through the local APIC, the APIC indicates the receipt of an illegal vector.

### 6.3.3 Software-Generated Interrupts

The `INT n` instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand. For example, the `INT 35` instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. If the processor's predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. If vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor's NMI-handling hardware is not activated.

Interrupts generated in software with the `INT n` instruction cannot be masked by the IF flag in the EFLAGS register.

## 6.4 SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

### 6.4.1 Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor-detectable exception. Exceptions are classified as **faults**, **traps**, and **aborts** (see Section 6.5, "Exception Classifications").

### 6.4.2 Software-Generated Exceptions

The `INTO`, `INT1`, `INT3`, and `BOUND` instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, `INT3` causes a breakpoint exception to be generated.

The `INT n` instruction can be used to emulate exceptions in software; but there is a limitation.<sup>1</sup> If `INT n` provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

### 6.4.3 Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation dependent. When a machine-check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code.

See Chapter 6, "Interrupt 18—Machine-Check Exception (#MC)" and Chapter 15, "Machine-Check Architecture," for more information about the machine-check mechanism.

---

1. The `INT n` instruction has opcode `CD` following by an immediate byte encoding the value of *n*. In contrast, `INT1` has opcode `F1` and `INT3` has opcode `CC`.

## 6.5 EXCEPTION CLASSIFICATIONS

Exceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

### NOTE

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. In this situation, the exception handler sees that the instruction pointer (CS:EIP) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

## 6.6 PROGRAM OR TASK RESTART

To allow the restarting of program or task following the handling of an exception or an interrupt, all exceptions (except aborts) are guaranteed to report exceptions on an instruction boundary. All interrupts are guaranteed to be taken on an instruction boundary.

For fault-class exceptions, the return instruction pointer (saved when the processor generates an exception) points to the faulting instruction. So, when a program or task is restarted following the handling of a fault, the faulting instruction is restarted (re-executed). Restarting the faulting instruction is commonly used to handle exceptions that are generated when access to an operand is blocked. The most common example of this type of fault is a page-fault exception (#PF) that occurs when a program or task references an operand located on a page that is not in memory. When a page-fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To insure that the restart is handled transparently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

For trap-class exceptions, the return instruction pointer points to the instruction following the trapping instruction. If a trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction. All trap exceptions allow program or task restart with no loss of continuity. For example, the overflow exception is a trap exception. Here, the return instruction pointer points to the instruction following the INTO instruction that tested EFLAGS.OF (overflow) flag. The trap handler for this exception resolves the overflow condition. Upon return from the trap handler, program or task execution continues at the instruction following the INTO instruction.

The abort-class exceptions do not support reliable restarting of the program or task. Abort handlers are designed to collect diagnostic information about the state of the processor when the abort exception occurred and then shut down the application and system as gracefully as possible.

Interrupts rigorously support restarting of interrupted programs and tasks without loss of continuity. The return instruction pointer saved for an interrupt points to the next instruction to be executed at the instruction boundary where the processor took the interrupt. If the instruction just executed has a repeat prefix, the interrupt is taken at the end of the current iteration with the registers set to execute the next iteration.

The ability of a P6 family processor to speculatively execute instructions does not affect the taking of interrupts by the processor. Interrupts are taken at instruction boundaries located during the retirement phase of instruction execution; so they are always taken in the “in-order” instruction stream. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the P6 family processors’ microarchitecture and its support for out-of-order instruction execution.

Note that the Pentium processor and earlier IA-32 processors also perform varying amounts of prefetching and preliminary decoding. With these processors as well, exceptions and interrupts are not signaled until actual “in-order” execution of the instructions. For a given code sample, the signaling of exceptions occurs uniformly when the code is executed on any family of IA-32 processors (except where new exceptions or new opcodes have been defined).

## 6.7 NONMASKABLE INTERRUPT (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors) with a delivery mode NMI.

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to insure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, “Handling Multiple NMIs”).

Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register.

It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor’s NMI-handling hardware can only be delivered through one of the mechanisms listed above.

### 6.7.1 Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMIs until the next execution of the IRET instruction. This blocking of NMIs prevents nested execution of the NMI handler. It is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts (see Section 6.8.1, “Masking Maskable Hardware Interrupts”).

An execution of the IRET instruction unblocks NMIs even if the instruction causes a fault. For example, if the IRET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated (see Section 20.2.7, “Sensitive Instructions”). In such a case, NMIs are unmasked before the exception handler is invoked.

## 6.8 ENABLING AND DISABLING INTERRUPTS

The processor inhibits the generation of some interrupts, depending on the state of the processor and of the IF and RF flags in the EFLAGS register, as described in the following sections.

### 6.8.1 Masking Maskable Hardware Interrupts

The IF flag can disable the servicing of maskable hardware interrupts received on the processor’s INTR pin or through the local APIC (see Section 6.3.2, “Maskable Hardware Interrupts”). When the IF flag is clear, the processor inhibits interrupts delivered to the INTR pin or through the local APIC from generating an internal interrupt request; when the IF flag is set, interrupts delivered to the INTR or through the local APIC pin are processed as normal external interrupts.

The IF flag does not affect non-maskable interrupts (NMIs) delivered to the NMI pin or delivery mode NMI messages delivered through the local APIC, nor does it affect processor generated exceptions. As with the other flags in the EFLAGS register, the processor clears the IF flag in response to a hardware reset.

The fact that the group of maskable hardware interrupts includes the reserved interrupt and exception vectors 0 through 32 can potentially cause confusion. Architecturally, when the IF flag is set, an interrupt for any of the vectors from 0 through 32 can be delivered to the processor through the INTR pin and any of the vectors from 16 through 32 can be delivered through the local APIC. The processor will then generate an interrupt and call the interrupt or exception handler pointed to by the vector number. So for example, it is possible to invoke the page-fault handler through the INTR pin (by means of vector 14); however, this is not a true page-fault exception. It is an interrupt. As with the INT  $n$  instruction (see Section 6.4.2, “Software-Generated Exceptions”), when an interrupt is generated through the INTR pin to an exception vector, the processor does not push an error code on the stack, so the exception handler may not operate correctly.

The IF flag can be set or cleared with the STI (set interrupt-enable flag) and CLI (clear interrupt-enable flag) instructions, respectively. These instructions may be executed only if the CPL is equal to or less than the IOPL. A general-protection exception (#GP) is generated if they are executed when the CPL is greater than the IOPL.<sup>2</sup> If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI.<sup>3</sup> The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag is also affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified flags back into the EFLAGS register.
- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is handled through an interrupt gate, the IF flag is automatically cleared, which disables maskable hardware interrupts. (If an interrupt is handled through a trap gate, the IF flag is not cleared.)

See the descriptions of the CLI, STI, PUSHF, POPF, and IRET instructions in Chapter 3, “Instruction Set Reference, A-L,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, and Chapter 4, “Instruction Set Reference, M-U,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a detailed description of the operations these instructions are allowed to perform on the IF flag.

## 6.8.2 Masking Instruction Breakpoints

The RF (resume) flag in the EFLAGS register controls the response of the processor to instruction-breakpoint conditions (see the description of the RF flag in Section 2.3, “System Flags and Fields in the EFLAGS Register”).

When set, it prevents an instruction breakpoint from generating a debug exception (#DB); when clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to prevent the processor from going into a debug exception loop on an instruction-breakpoint. See Section 17.3.1.1, “Instruction-Breakpoint Exception Condition,” for more information on the use of this flag.

As noted in Section 6.8.3, execution of the MOV or POP instruction to load the SS register suppresses any instruction breakpoint on the next instruction (just as if EFLAGS.RF were 1).

## 6.8.3 Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, StackTop
```

(Software might also use the POP instruction to load SS and ESP.)

2. The effect of the IOPL on these instructions is modified slightly when the virtual mode extension is enabled by setting the VME flag in control register CR4: see Section 20.3, “Interrupt and Exception Handling in Virtual-8086 Mode.” Behavior is also impacted by the PVI flag: see Section 20.4, “Protected-Mode Virtual Interrupts.”
3. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

To account for this situation, the processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction. The following items provide details:

- Any instruction breakpoint on the next instruction is suppressed (as if EFLAGS.RF were 1).
- Any data breakpoint on the MOV to SS instruction or POP to SS instruction is inhibited until the instruction boundary following the next instruction.
- Any single-step trap that would be delivered following the MOV to SS instruction or POP to SS instruction (because EFLAGS.TF is 1) is suppressed.
- The suppression and inhibition ends after delivery of an exception or the execution of the next instruction.
- If a sequence of consecutive instructions each loads the SS register (using MOV or POP), only the first is guaranteed to inhibit or suppress events in this way.

Intel recommends that software use the LSS instruction to load the SS register and ESP together. The problem identified earlier does not apply to LSS, and the LSS instruction does not inhibit events as detailed above.

## 6.9 PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS

If more than one exception or interrupt is pending at an instruction boundary, the processor services them in a predictable order. Table 6-2 shows the priority among classes of exception and interrupt sources.

**Table 6-2. Priority Among Simultaneous Exceptions and Interrupts**

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) <sup>1</sup>
6	Maskable Hardware Interrupts <sup>1</sup>
7	Code Breakpoint Fault
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault

**Table 6-2. Priority Among Simultaneous Exceptions and Interrupts (Contd.)**

9	Faults from Decoding the Next Instruction <ul style="list-style-type: none"> <li>- Instruction length &gt; 15 bytes</li> <li>- Invalid Opcode</li> <li>- Coprocessor Not Available</li> </ul>
10 (Lowest)	Faults on Executing an Instruction <ul style="list-style-type: none"> <li>- Overflow</li> <li>- Bound error</li> <li>- Invalid TSS</li> <li>- Segment Not Present</li> <li>- Stack fault</li> <li>- General Protection</li> <li>- Data Page Fault</li> <li>- Alignment Check</li> <li>- x87 FPU Floating-point exception</li> <li>- SIMD floating-point exception</li> <li>- Virtualization exception</li> </ul>

**NOTE**

1. The Intel® 486 processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

While priority among these classes listed in Table 6-2 is consistent throughout the architecture, exceptions within each class are implementation-dependent and may vary from processor to processor. The processor first services a pending exception or interrupt from the class which has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions are re-generated when the interrupt handler returns execution to the point in the program or task where the exceptions and/or interrupts occurred.

## 6.10 INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors (in protected mode). Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the IDT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is,  $8N - 1$ ).

The IDT may reside anywhere in the linear address space. As shown in Figure 6-1, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SIDT (store IDT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another. The SIDT instruction copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

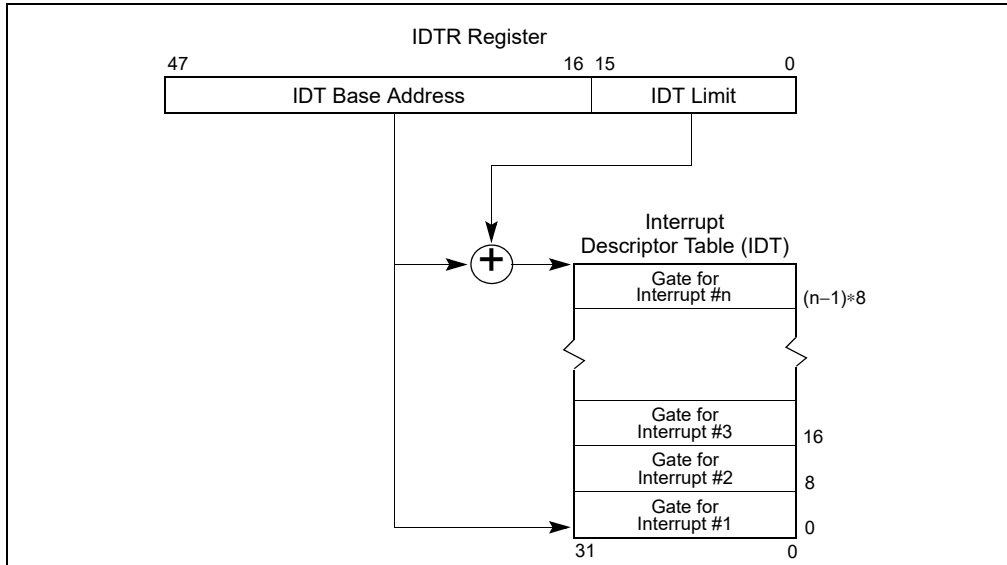
If a vector references a descriptor beyond the limit of the IDT, a general-protection exception (#GP) is generated.



**NOTE**

Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery.

IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. The same apply for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.



**Figure 6-1. Relationship of the IDTR and IDT**

## 6.11 IDT DESCRIPTORS

The IDT may contain any of three kinds of gate descriptors:

- Task-gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Figure 6-2 shows the formats for the task-gate, interrupt-gate, and trap-gate descriptors. The format of a task gate used in an IDT is the same as that of a task gate used in the GDT or an LDT (see Section 7.2.5, “Task-Gate Descriptor”). The task gate contains the segment selector for a TSS for an exception and/or interrupt handler task.

Interrupt and trap gates are very similar to call gates (see Section 5.8.3, “Call Gates”). They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt-handler code segment. These gates differ in the way the processor handles the IF flag in the EFLAGS register (see Section 6.12.1.2, “Flag Usage By Exception- or Interrupt-Handler Procedure”).

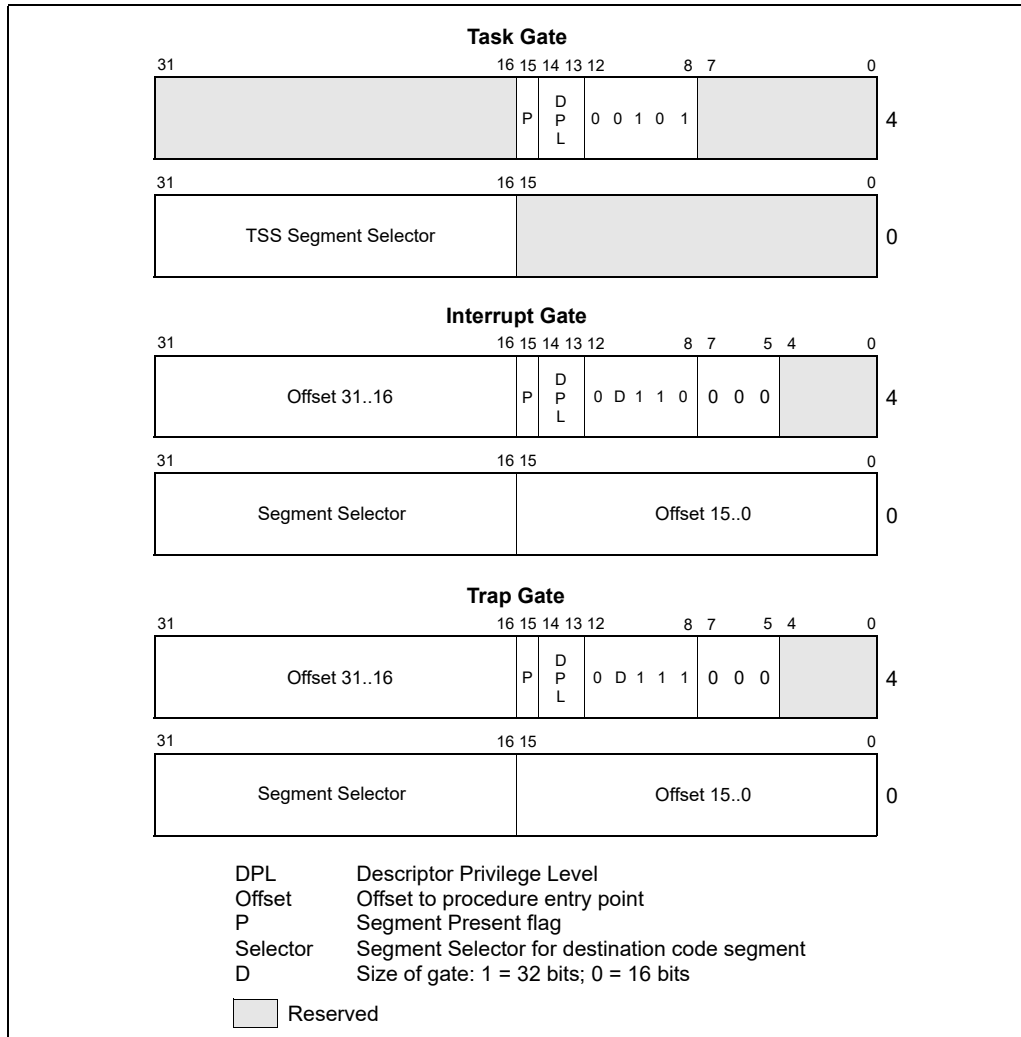


Figure 6-2. IDT Gate Descriptors

## 6.12 EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 7.3, "Task Switching").

### 6.12.1 Exception- or Interrupt-Handler Procedures

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task (see Figure 6-3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt-handling procedure.

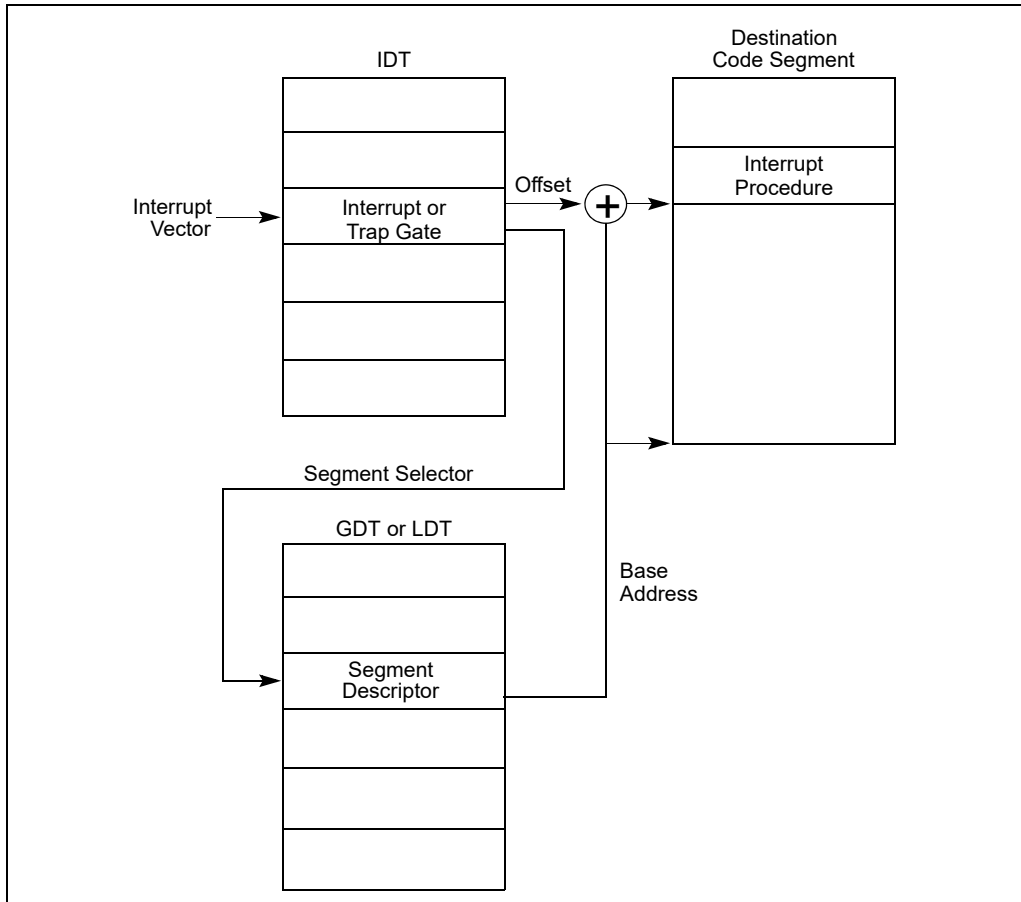


Figure 6-3. Interrupt Procedure Call

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  - a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  - b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figures 6-4).
  - c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  - a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figures 6-4).
  - b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

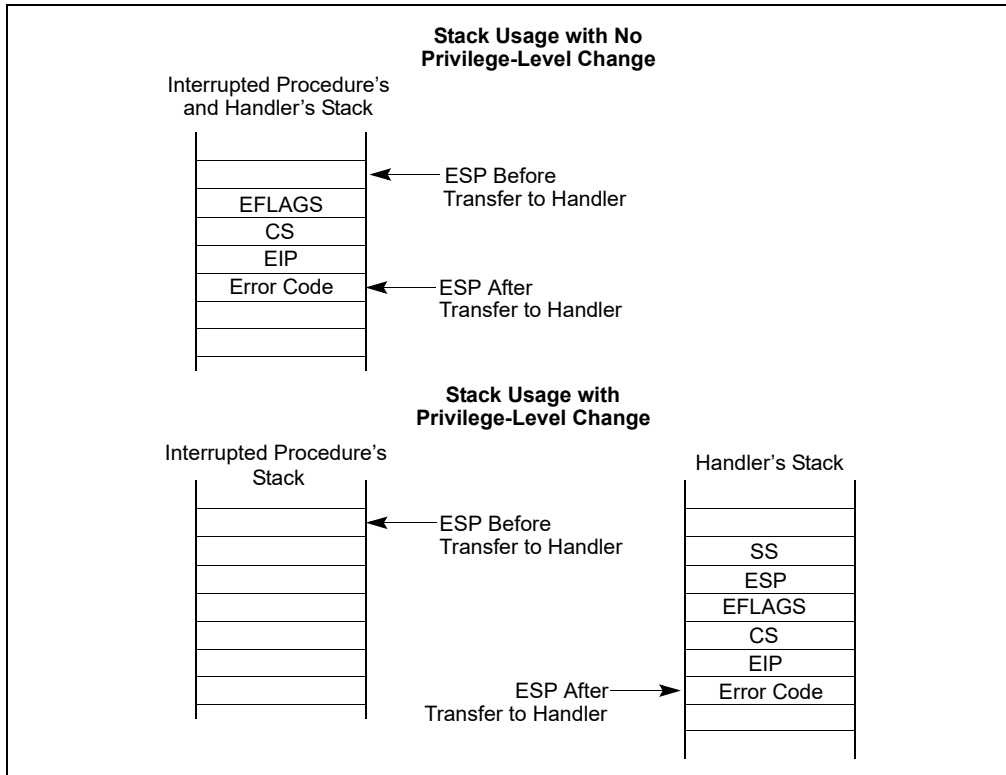


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for a description of the complete operation performed by the IRET instruction.

If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

### 6.12.1.1 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 5.8.4, "Accessing a Code Segment Through a Call Gate"). The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general-protection exception (#GP). The protection mechanism for exception- and interrupt-handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.
- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an INT *n*, INT3, or INTO instruction.<sup>4</sup> Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are

4. This check is not performed by execution of the INT1 instruction (opcode F1); it would be performed by execution of INT 1 (opcode CD 01).

placed in more privileged code segments (numerically lower privilege level). For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt-handling procedures can run. Either of the following techniques can be used to avoid privilege-level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

### 6.12.1.2 Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response and ensures that no single-step exception will be delivered after delivery to the handler. A subsequent IRET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register. When accessing an exception- or interrupt-handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

## 6.12.2 Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

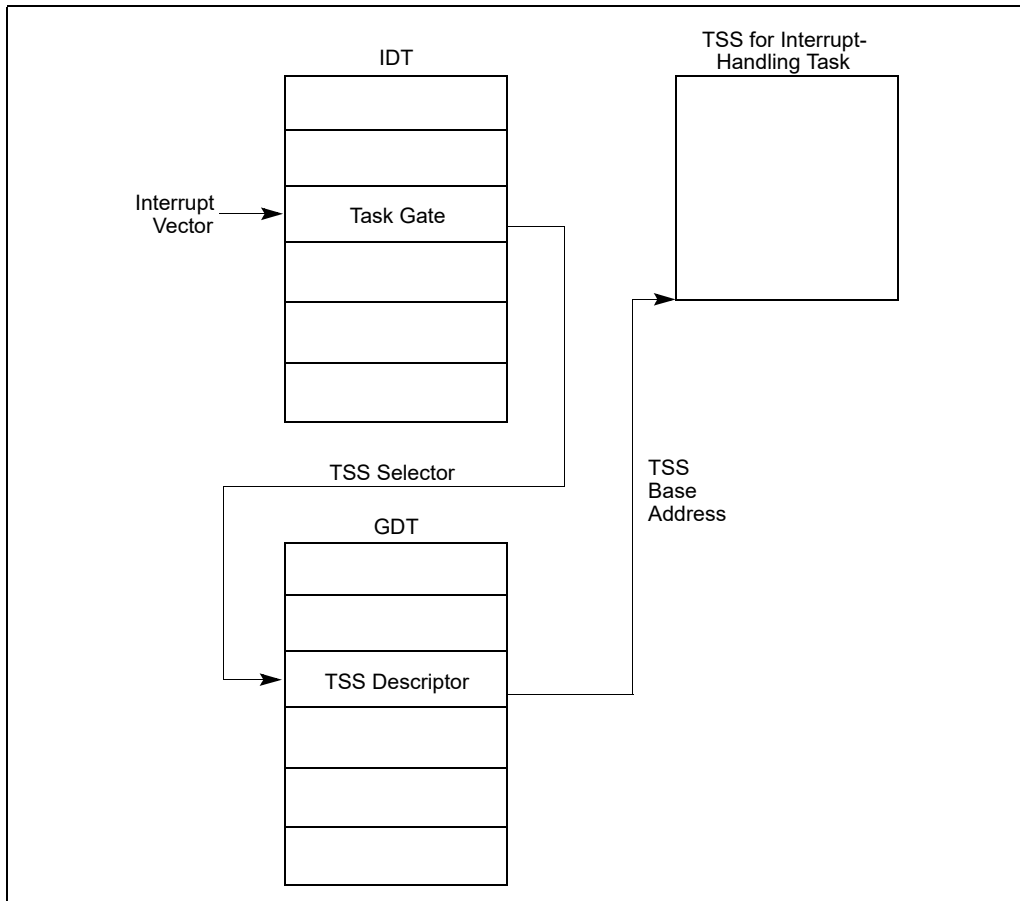
The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency.

A task gate in the IDT references a TSS descriptor in the GDT (see Figure 6-5). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 7.3, "Task Switching"). The link back to the interrupted task is stored in the previous task link field of the handler task's TSS. If an exception caused an error code to be generated, this error code is copied to the stack of the new task.

When exception- or interrupt-handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism). The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.

**NOTE**

Because IA-32 architecture tasks are not re-entrant, an interrupt-handler task must disable interrupts between the time it completes handling the interrupt and the time it executes the IRET instruction. This action prevents another interrupt from occurring while the interrupt task's TSS is still marked busy, which would cause a general-protection (#GP) exception.



**Figure 6-5. Interrupt Task Switch**

### 6.13 ERROR CODE

When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure 6-6. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

- EXT**      **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.<sup>5</sup> The bit is cleared if the exception occurred during delivery of a software interrupt (INT<sub>n</sub>, INT3, or INTO).
- IDT**      **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.
- TI**        **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

<sup>5</sup> The bit is also set if the exception occurred during delivery of INT1.

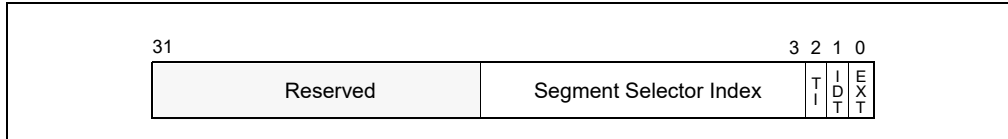


Figure 6-6. Error Code

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment selector was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the “Interrupt 14—Page-Fault Exception (#PF)” section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INT *n* instruction, even if an error code is normally produced for those exceptions.

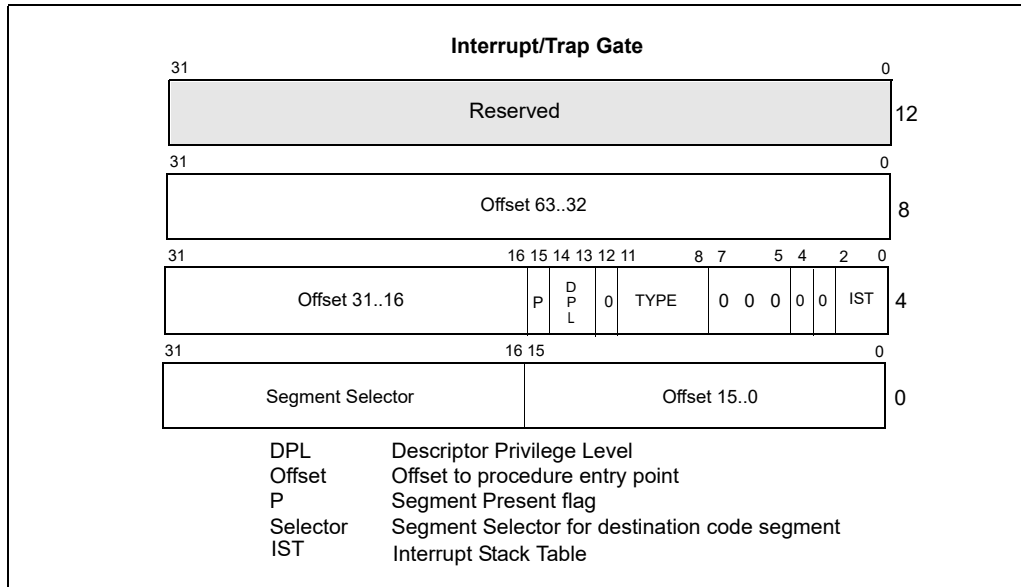
## 6.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE

In 64-bit mode, interrupt and exception handling is similar to what has been described for non-64-bit modes. The following are the exceptions:

- All interrupt handlers pointed by the IDT are in 64-bit code (this does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits; and the processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy modes, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism.
- The alignment of interrupt stack frame is different.

### 6.14.1 64-Bit Mode IDT

Interrupt and trap gates are 16 bytes in length to provide a 64-bit offset for the instruction pointer (RIP). The 64-bit RIP referenced by interrupt-gate descriptors allows an interrupt service routine to be located anywhere in the linear-address space. See Figure 6-7.



**Figure 6-7. 64-Bit IDT Gate Descriptors**

In 64-bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7:0) of a 64-bit mode interrupt gate are similar but not identical to legacy 32-bit interrupt gates. The type field (bits 11:8 in bytes 7:4) is described in Table 3-2. The Interrupt Stack Table (IST) field (bits 4:0 in bytes 7:4) is used by the stack switching mechanisms described in Section 6.14.5, "Interrupt Stack Table." Bytes 11:8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general-protection exception (#GP) is generated if software attempts to reference an interrupt gate with a target RIP that is not in canonical form.

The target code segment referenced by the interrupt gate must be a 64-bit code segment (CS.L = 1, CS.D = 0). If the target is not a 64-bit code segment, a general-protection exception (#GP) is generated with the IDT vector number reported as the error code.

Only 64-bit interrupt and trap gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). Legacy 32-bit interrupt or trap gate types (0EH or 0FH) are redefined in IA-32e mode as 64-bit interrupt and trap gate types. No 32-bit interrupt or trap gate type exists in IA-32e mode. If a reference is made to a 16-bit interrupt or trap gate (06H or 07H), a general-protection exception (#GP(0)) is generated.

### 6.14.2 64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes. SS:ESP is pushed only on a CPL change. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at eight bytes. This is because only 64-bit mode gates can be referenced. 64-bit mode also pushes SS:RSP unconditionally, rather than only on a CPL change.

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stackframe size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an additional error code place-holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA-32e mode, the RSP is aligned to a 16-byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16-byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent IRET.

Aligning the stack permits exception and interrupt frames to be aligned on a 16-byte boundary before interrupts are re-enabled. This allows the stack to be formatted for optimal storage of 16-byte XMM registers, which enables



the interrupt handler to use faster 16-byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel-mode case where there is no stack switch or IST used. For a stack switch or IST, the OS would have presumably put suitably aligned RSP values in the TSS.

### 6.14.3 IRET in IA-32e Mode

In IA-32e mode, IRET executes with an 8-byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8-byte IRET operand size works correctly.

Because interrupt stack-frame pushes are always eight bytes in IA-32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ESP/RSP size adjustment is determined by the stack size.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL = 0. This is because the original interrupt always pushes SS:RSP.

In IA-32e mode, IRET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL  $\neq$  3, IRET allows SS to be loaded with a NULL selector. As part of the stack switch mechanism, an interrupt or exception sets the new SS to NULL, instead of fetching a new SS selector from the TSS and loading the corresponding descriptor from the GDT or LDT. The new SS selector is set to NULL in order to properly handle returns from subsequent nested far transfers. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent IRET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

### 6.14.4 Stack Switching in IA-32e Mode

The IA-32 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The 64-bit extensions of Intel 64 architecture implement a modified version of the legacy stack-switching mechanism and an alternative stack-switching mechanism called the interrupt stack table (IST).

In IA-32 modes, the legacy IA-32 stack-switch mechanism is unchanged. In IA-32e mode, the legacy stack-switch mechanism is modified. When stacks are switched as part of a 64-bit mode privilege-level change (resulting from an interrupt), a new SS descriptor is not loaded. IA-32e mode loads only an inner-level RSP from the TSS. The new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INT, interrupts and exceptions). The old SS and RSP are saved on the new stack (Figure 6-8). On the subsequent IRET, the old SS is popped from the stack and loaded into the SS register.

In summary, a stack switch in IA-32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.

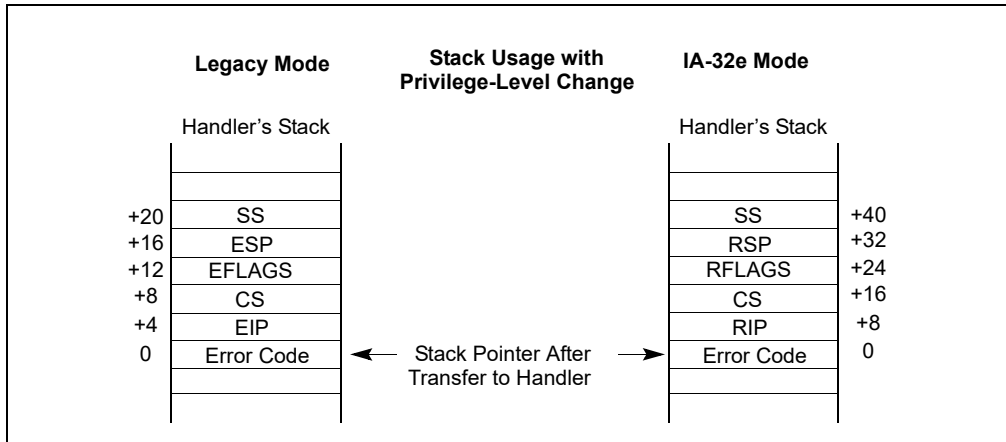


Figure 6-8. IA-32e Mode Stack Usage After Privilege Level Change

### 6.14.5 Interrupt Stack Table

In IA-32e mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism described above. This mechanism unconditionally switches stacks when it is enabled. It can be enabled on an individual interrupt-vector basis using a field in the IDT entry. This means that some interrupt vectors can use the modified legacy mechanism and others can use the IST mechanism.

The IST mechanism is only available in IA-32e mode. It is part of the 64-bit mode TSS. The motivation for the IST mechanism is to provide a method for specific interrupts (such as NMI, double-fault, and machine-check) to always execute on a known good stack. In legacy mode, interrupts can use the task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the legacy task-switch mechanism is not supported in IA-32e mode.

The IST mechanism provides up to seven IST pointers in the TSS. The pointers are referenced by an interrupt-gate descriptor in the interrupt-descriptor table (IDT); see Figure 6-7. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

## 6.15 EXCEPTION AND INTERRUPT REFERENCE

The following sections describe conditions which generate exceptions and interrupts. They are arranged in the order of vector numbers. The information contained in these sections are as follows:

- **Exception Class** — Indicates whether the exception class is a fault, trap, or abort type. Some exceptions can be either a fault or trap type, depending on when the error condition is detected. (This section is not applicable to interrupts.)
- **Description** — Gives a general description of the purpose of the exception or interrupt type. It also describes how the processor handles the exception or interrupt.
- **Exception Error Code** — Indicates whether an error code is saved for the exception. If one is saved, the contents of the error code are described. (This section is not applicable to interrupts.)
- **Saved Instruction Pointer** — Describes which instruction the saved (or return) instruction pointer points to. It also indicates whether the pointer can be used to restart a faulting instruction.
- **Program State Change** — Describes the effects of the exception or interrupt on the state of the currently running program or task and the possibilities of restarting the program or task without loss of continuity.

## Interrupt 0—Divide Error Exception (#DE)

**Exception Class**     **Fault.**

### Description

Indicates the divisor operand for a DIV or IDIV instruction is 0 or that the result cannot be represented in the number of bits specified for the destination operand.

### Exception Error Code

None.

### Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany the divide error, because the exception occurs before the faulting instruction is executed.

## Interrupt 1—Debug Exception (#DB)

**Exception Class**      **Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.**

### Description

Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 6-3). See Chapter 17, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for detailed information about the debug exceptions.

**Table 6-3. Debug Exception Conditions and Corresponding Exception Classes**

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap
Execution of INT1 <sup>1</sup>	Trap

### NOTES:

1. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

### Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

### Saved Instruction Pointer

Fault — Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap — Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

### Program State Change

Fault — A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap — A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

The following items detail the treatment of debug exceptions on the instruction boundary following execution of the MOV or the POP instruction that loads the SS register:

- If EFLAGS.TF is 1, no single-step trap is generated.
- If the instruction encounters a data breakpoint, the resulting debug exception is delivered after completion of the instruction after the MOV or POP. This occurs even if the next instruction is INT *n*, INT3, or INTO.
- Any instruction breakpoint on the instruction after the MOV or POP is suppressed (as if EFLAGS.RF were 1).

Any debug exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a debug exception instead causes execution to roll back to just before the XBEGIN instruction

and then delivers a #DB. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## Interrupt 2—NMI Interrupt

**Exception Class**     **Not applicable.**

### Description

The nonmaskable interrupt (NMI) is generated externally by asserting the processor's NMI pin or through an NMI request set by the I/O APIC to the local APIC. This interrupt causes the NMI interrupt handler to be called.

### Exception Error Code

Not applicable.

### Saved Instruction Pointer

The processor always takes an NMI interrupt on an instruction boundary. The saved contents of CS and EIP registers point to the next instruction to be executed at the point the interrupt is taken. See Section 6.5, "Exception Classifications," for more information about when the processor takes NMI interrupts.

### Program State Change

The instruction executing when an NMI interrupt is received is completed before the NMI is generated. A program or task can thus be restarted upon returning from an interrupt handler without loss of continuity, provided the interrupt handler saves the state of the processor before handling the interrupt and restores the processor's state prior to a return.

## Interrupt 3—Breakpoint Exception (#BP)

**Exception Class**    **Trap.**

### Description

Indicates that a breakpoint instruction (INT3, opcode CC) was executed, causing a breakpoint trap to be generated. Typically, a debugger sets a breakpoint by replacing the first opcode byte of an instruction with the opcode for the INT3 instruction. (The INT3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode.) The operating system or a debugging tool can use a data segment mapped to the same physical address space as the code segment to place an INT3 instruction in places where it is desired to call the debugger.

With the P6 family, Pentium, Intel486, and Intel386 processors, it is more convenient to set breakpoints with the debug registers. (See Section 17.3.2, “Breakpoint Exception (#BP)—Interrupt Vector 3,” for information about the breakpoint exception.) If more breakpoints are needed beyond what the debug registers allow, the INT3 instruction can be used.

Any breakpoint exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a break exception instead causes execution to roll back to just before the XBEGIN instruction and then delivers a **debug exception (#DB)** — **not** a breakpoint exception. See Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

A breakpoint exception can also be generated by executing the INT *n* instruction with an operand of 3. The action of this instruction (INT 3) is slightly different than that of the INT3 instruction (see “INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

### Exception Error Code

None.

### Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction following the INT3 instruction.

### Program State Change

Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. The debugger can thus resume the suspended program by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the saved contents of the EIP register. Upon returning from the debugger, program execution resumes with the replaced instruction.

## Interrupt 4—Overflow Exception (#OF)

**Exception Class**      **Trap.**

### Description

Indicates that an overflow trap occurred when an INTO instruction was executed. The INTO instruction checks the state of the OF flag in the EFLAGS register. If the OF flag is set, an overflow trap is generated.

Some arithmetic instructions (such as the ADD and SUB) perform both signed and unsigned arithmetic. These instructions set the OF and CF flags in the EFLAGS register to indicate signed overflow and unsigned overflow, respectively. When performing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used. The benefit of using the INTO instruction is that if the overflow exception is detected, an exception handler can be called automatically to handle the overflow condition.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction following the INTO instruction.

### Program State Change

Even though the EIP points to the instruction following the INTO instruction, the state of the program is essentially unchanged because the INTO instruction does not affect any register or memory locations. The program can thus resume normal execution upon returning from the overflow exception handler.



## Interrupt 5—BOUND Range Exceeded Exception (#BR)

**Exception Class**     **Fault.**

### Description

Indicates that a BOUND-range-exceeded fault occurred when a BOUND instruction was executed. The BOUND instruction checks that a signed array index is within the upper and lower bounds of an array located in memory. If the array index is not within the bounds of the array, a BOUND-range-exceeded fault is generated.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the BOUND instruction that generated the exception.

### Program State Change

A program-state change does not accompany the bounds-check fault, because the operands for the BOUND instruction are not modified. Returning from the BOUND-range-exceeded exception handler causes the BOUND instruction to be restarted.

## Interrupt 6—Invalid Opcode Exception (#UD)

**Exception Class**     **Fault.**

### Description

Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMA $SW$ , PMA $XUB$ , PMIN $SW$ , PMIN $UB$ , PMOVMSKB, PMULHUW, PSADBW, PSHUFW, PADDQ, PSUBQ, PALIGNR, PABS $B$ , PABS $D$ , PABS $W$ , PHADD $D$ , PHADD $SW$ , PHADD $W$ , PHSUB $D$ , PHSUB $SW$ , PHSUB $W$ , PMADDUBSM, PMULHR $SW$ , PSHUFB, PSIGN $B$ , PSIGN $D$ , and PSIGN $W$ .
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD0, UD1 or UD2 instruction. Note that even though it is the execution of the UD0, UD1 or UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still points at the UD0, UD1 or UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, "Exception Classifications," for general rules for taking of interrupts and exceptions.)

The opcodes D6 and F1 are undefined opcodes reserved by the Intel 64 and IA-32 architectures. These opcodes, even though undefined, do not generate an invalid opcode exception.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany an invalid-opcode fault, because the invalid instruction is not executed.

## Interrupt 7—Device Not Available Exception (#NM)

**Exception Class**     **Fault.**

### Description

Indicates one of the following things:

The device-not-available exception is generated by either of three conditions:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register CR0 was set (1). See the paragraph below for the special case of the WAIT/FWAIT instruction.
- The processor executed a WAIT/FWAIT instruction while the MP and TS flags of register CR0 were set, regardless of the setting of the EM flag.
- The processor executed an x87 FPU, MMX, or SSE/SSE2/SSE3 instruction (with the exception of MOVNTI, PAUSE, PREFETCH $h$ , SFENCE, LFENCE, MFENCE, and CLFLUSH) while the TS flag in control register CR0 was set and the EM flag is clear.

The EM flag is set when the processor does not have an internal x87 FPU floating-point unit. A device-not-available exception is then generated each time an x87 FPU floating-point instruction is encountered, allowing an exception handler to call floating-point instruction emulation routines.

The TS flag indicates that a context switch (task switch) has occurred since the last time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction was executed; but that the context of the x87 FPU, XMM, and MXCSR registers were not saved. When the TS flag is set and the EM flag is clear, the processor generates a device-not-available exception each time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction is encountered (with the exception of the instructions listed above). The exception handler can then save the context of the x87 FPU, XMM, and MXCSR registers before it executes the instruction. See Section 2.5, "Control Registers," for more information about the TS flag.

The MP flag in control register CR0 is used along with the TS flag to determine if WAIT or FWAIT instructions should generate a device-not-available exception. It extends the function of the TS flag to the WAIT and FWAIT instructions, giving the exception handler an opportunity to save the context of the x87 FPU before the WAIT or FWAIT instruction is executed. The MP flag is provided primarily for use with the Intel 286 and Intel386 DX processors. For programs running on the Pentium 4, Intel Xeon, P6 family, Pentium, or Intel486 DX processors, or the Intel 487 SX coprocessors, the MP flag should always be set; for programs running on the Intel486 SX processor, the MP flag should be clear.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point instruction or the WAIT/FWAIT instruction that generated the exception.

### Program State Change

A program-state change does not accompany a device-not-available fault, because the instruction that generated the exception is not executed.

If the EM flag is set, the exception handler can then read the floating-point instruction pointed to by the EIP and call the appropriate emulation routine.

If the MP and TS flags are set or the TS flag alone is set, the exception handler can save the context of the x87 FPU, clear the TS flag, and continue execution at the interrupted floating-point or WAIT/FWAIT instruction.

## Interrupt 8—Double Fault Exception (#DF)

**Exception Class**    **Abort.**

### Description

Indicates that the processor detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception. To determine when two faults need to be signalled as a double fault, the processor divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults (see Table 6-4).

**Table 6-4. Interrupt and Exception Classes**

Class	Vector Number	Description
Benign Exceptions and Interrupts	1	Debug
	2	NMI Interrupt
	3	Breakpoint
	4	Overflow
	5	BOUND Range Exceeded
	6	Invalid Opcode
	7	Device Not Available
	9	Coprocessor Segment Overrun
	16	Floating-Point Error
	17	Alignment Check
	18	Machine Check
	19	SIMD floating-point
	All	INT <i>n</i>
All	INTR	
Contributory Exceptions	0	Divide Error
	10	Invalid TSS
	11	Segment Not Present
	12	Stack Fault
	13	General Protection
Page Faults	14	Page Fault
	20	Virtualization Exception

Table 6-5 shows the various combinations of exception classes that cause a double fault to be generated. A double-fault exception falls in the abort class of exceptions. The program or task cannot be restarted or resumed. The double-fault handler can be used to collect diagnostic information about the state of the machine and/or, when possible, to shut the application and/or system down gracefully or restart the system.

A segment or page fault may be encountered while prefetching instructions; however, this behavior is outside the domain of Table 6-5. Any further faults generated while the processor is attempting to transfer control to the appropriate fault handler could still lead to a double-fault sequence.

**Table 6-5. Conditions for Generating a Double Fault**

First Exception	Second Exception		
	Benign	Contributory	Page Fault
Benign	Handle Exceptions Serially	Handle Exceptions Serially	Handle Exceptions Serially
Contributory	Handle Exceptions Serially	Generate a Double Fault	Handle Exceptions Serially
Page Fault	Handle Exceptions Serially	Generate a Double Fault	Generate a Double Fault
Double Fault	Handle Exceptions Serially	Enter Shutdown Mode	Enter Shutdown Mode

If another contributory or page fault exception occurs while attempting to call the double-fault handler, the processor enters shutdown mode. This mode is similar to the state following execution of an HLT instruction. In this mode, the processor stops executing instructions until an NMI interrupt, SMI interrupt, hardware reset, or INIT# is received. The processor generates a special bus cycle to indicate that it has entered shutdown mode. Software designers may need to be aware of the response of hardware when it goes into shutdown mode. For example, hardware may turn on an indicator light on the front panel, generate an NMI interrupt to record diagnostic information, invoke reset initialization, generate an INIT initialization, or generate an SMI. If any events are pending during shutdown, they will be handled after a wake event from shutdown is processed (for example, A20M# interrupts).

If a shutdown occurs while the processor is executing an NMI interrupt handler, then only a hardware reset can restart the processor. Likewise, if the shutdown occurs while executing in SMM, a hardware reset must be used to restart the processor.

**Exception Error Code**

Zero. The processor always pushes an error code of 0 onto the stack of the double-fault handler.

**Saved Instruction Pointer**

The saved contents of CS and EIP registers are undefined.

**Program State Change**

A program-state following a double-fault exception is undefined. The program or task cannot be resumed or restarted. The only available action of the double-fault exception handler is to collect all possible context information for use in diagnostics and then close the application and/or shut down or reset the processor.

If the double fault occurs when any portion of the exception handling machine state is corrupted, the handler cannot be invoked and the processor must be reset.

## Interrupt 9—Coprocessor Segment Overrun

**Exception Class**     **Abort. (Intel reserved; do not use. Recent IA-32 processors do not generate this exception.)**

### Description

Indicates that an Intel386 CPU-based systems with an Intel 387 math coprocessor detected a page or segment violation while transferring the middle portion of an Intel 387 math coprocessor operand. The P6 family, Pentium, and Intel486 processors do not generate this exception; instead, this condition is detected with a general protection exception (#GP), interrupt 13.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state following a coprocessor segment-overrun exception is undefined. The program or task cannot be resumed or restarted. The only available action of the exception handler is to save the instruction pointer and reinitialize the x87 FPU using the FNINIT instruction.

## Interrupt 10—Invalid TSS Exception (#TS)

**Exception Class**     **Fault.**

### Description

Indicates that there was an error related to a TSS. Such an error might be detected during a task switch or during the execution of instructions that use information from a TSS. Table 6-6 shows the conditions that cause an invalid TSS exception to be generated.

**Table 6-6. Invalid TSS Conditions**

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment limit is less than 67H for 32-bit TSS or less than 2CH for 16-bit TSS.
TSS segment selector index	During an IRET task switch, the TI flag in the TSS segment selector indicates the LDT.
TSS segment selector index	During an IRET task switch, the TSS segment selector exceeds descriptor table limit.
TSS segment selector index	During an IRET task switch, the busy flag in the TSS descriptor indicates an inactive task.
TSS segment selector index	During a task switch, an attempt to access data in a TSS results in a limit violation or canonical fault.
TSS segment selector index	During an IRET task switch, the backlink is a NULL selector.
TSS segment selector index	During an IRET task switch, the backlink points to a descriptor which is not a busy TSS.
TSS segment selector index	The new TSS descriptor is beyond the GDT limit.
TSS segment selector index	The new TSS selector is null on an attempt to lock the new TSS.
TSS segment selector index	The new TSS selector has the TI bit set on an attempt to lock the new TSS.
TSS segment selector index	The new TSS descriptor is not an available TSS descriptor on an attempt to lock the new TSS.
LDT segment selector index	LDT not valid or not present.
Stack segment selector index	The stack segment selector exceeds descriptor table limit.
Stack segment selector index	The stack segment selector is NULL.
Stack segment selector index	The stack segment descriptor is a non-data segment.
Stack segment selector index	The stack segment is not writable.
Stack segment selector index	The stack segment DPL ≠ CPL.
Stack segment selector index	The stack segment selector RPL ≠ CPL.
Code segment selector index	The code segment selector exceeds descriptor table limit.
Code segment selector index	The code segment selector is NULL.
Code segment selector index	The code segment descriptor is not a code segment type.
Code segment selector index	The nonconforming code segment DPL ≠ CPL.
Code segment selector index	The conforming code segment DPL is greater than CPL.
Data segment selector index	The data segment selector exceeds the descriptor table limit.
Data segment selector index	The data segment descriptor is not a readable code or data type.
Data segment selector index	The data segment descriptor is a nonconforming code type and RPL > DPL.
Data segment selector index	The data segment descriptor is a nonconforming code type and CPL > DPL.
TSS segment selector index	The TSS segment descriptor/upper descriptor is beyond the GDT segment limit.
TSS segment selector index	The TSS segment descriptor is not an available TSS type.
TSS segment selector index	The TSS segment descriptor is an available 286 TSS type in IA-32e mode.

**Table 6-6. Invalid TSS Conditions (Contd.)**

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment upper descriptor is not the correct type.
TSS segment selector index	The TSS segment descriptor contains a non-canonical base.

This exception can be generated either in the context of the original task or in the context of the new task (see Section 7.3, "Task Switching"). Until the processor has completely verified the presence of the new TSS, the exception is generated in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete. Any invalid-TSS conditions detected after this point are handled in the context of the new task. (A task switch is considered complete when the task register is loaded with the segment selector for the new TSS and, if the switch is due to a procedure call or interrupt, the previous task link field of the new TSS references the old TSS.)

The invalid-TSS handler must be a task called using a task gate. Handling this exception inside the faulting TSS context is not recommended because the processor state may not be consistent.

### Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception was caused by an event external to the currently running program (for example, if an external interrupt handler using a task gate attempted a task switch to an invalid TSS).

### Saved Instruction Pointer

If the exception condition was detected before the task switch was carried out, the saved contents of CS and EIP registers point to the instruction that invoked the task switch. If the exception condition was detected after the task switch was carried out, the saved contents of CS and EIP registers point to the first instruction of the new task.

### Program State Change

The ability of the invalid-TSS handler to recover from the fault depends on the error condition that causes the fault. See Section 7.3, "Task Switching," for more information on the task switch process and the possible recovery actions that can be taken.

If an invalid TSS exception occurs during a task switch, it can occur before or after the commit-to-new-task point. If it occurs before the commit point, no program state change occurs. If it occurs after the commit point (when the segment descriptor information for the new segment selectors have been loaded in the segment registers), the processor will load all the state information from the new TSS before it generates the exception. During a task switch, the processor first loads all the segment registers with segment selectors from the TSS, then checks their contents for validity. If an invalid TSS exception is discovered, the remaining segment registers are loaded but not checked for validity and therefore may not be usable for referencing memory. The invalid TSS handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should load all segment registers before trying to resume the new task; otherwise, general-protection exceptions (#GP) may result later under conditions that make diagnosis more difficult. The Intel recommended way of dealing with this situation is to use a task for the invalid TSS exception handler. The task switch back to the interrupted task from the invalid-TSS exception-handler task will then cause the processor to check the registers as it loads them from the TSS.



## Interrupt 11—Segment Not Present (#NP)

**Exception Class**     **Fault.**

### Description

Indicates that the present flag of a segment or gate descriptor is clear. The processor can generate this exception during any of the following operations:

- While attempting to load CS, DS, ES, FS, or GS registers. [Detection of a not-present segment while loading the SS register causes a stack fault exception (#SS) to be generated.] This situation can occur while performing a task switch.
- While attempting to load the LDTR using an LLDT instruction. Detection of a not-present LDT while loading the LDTR during a task switch operation causes an invalid-TSS exception (#TS) to be generated.
- When executing the LTR instruction and the TSS is marked not present.
- While attempting to use a gate descriptor or TSS that is marked segment-not-present, but is otherwise valid.

An operating system typically uses the segment-not-present exception to implement virtual memory at the segment level. If the exception handler loads the segment and returns, the interrupted program or task resumes execution.

A not-present indication in a gate descriptor, however, does not indicate that a segment is not present (because gates do not correspond to segments). The operating system may use the present flag for gate descriptors to trigger exceptions of special significance to the operating system.

A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

### Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

- an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment
- a benign exception that subsequently referenced a not-present segment

The IDT flag is set if the error code refers to an IDT entry. This occurs when the IDT entry for an interrupt being serviced references a not-present gate. Such an event could be generated by an INT instruction or a hardware interrupt.

### Saved Instruction Pointer

The saved contents of CS and EIP registers normally point to the instruction that generated the exception. If the exception occurred while loading segment descriptors for the segment selectors in a new TSS, the CS and EIP registers point to the first instruction in the new task. If the exception occurred while accessing a gate descriptor, the CS and EIP registers point to the instruction that invoked the access (for example a CALL instruction that references a call gate).

### Program State Change

If the segment-not-present exception occurs as the result of loading a register (CS, DS, SS, ES, FS, GS, or LDTR), a program-state change does accompany the exception because the register is not loaded. Recovery from this exception is possible by simply loading the missing segment into memory and setting the present flag in the segment descriptor.

If the segment-not-present exception occurs while accessing a gate descriptor, a program-state change does not accompany the exception. Recovery from this exception is possible merely by setting the present flag in the gate descriptor.

If a segment-not-present exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change

occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The segment-not-present exception handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

## Interrupt 12—Stack Fault Exception (#SS)

**Exception Class**     **Fault.**

### Description

Indicates that one of the following stack related conditions was detected:

- A limit violation is detected during an operation that refers to the SS register. Operations that can cause a limit violation include stack-oriented instructions such as POP, PUSH, CALL, RET, IRET, ENTER, and LEAVE, as well as other memory references which implicitly or explicitly use the SS register (for example, MOV AX, [BP+6] or MOV AX, SS:[EAX+6]). The ENTER instruction generates this exception when there is not enough stack space for allocating local variables.
- A not-present stack segment is detected when attempting to load the SS register. This violation can occur during the execution of a task switch, a CALL instruction to a different privilege level, a return to a different privilege level, an LSS instruction, or a MOV or POP instruction to the SS register.
- A canonical violation is detected in 64-bit mode during an operation that reference memory using the stack pointer register containing a non-canonical memory address.

Recovery from this fault is possible by either extending the limit of the stack segment (in the case of a limit violation) or loading the missing stack segment into memory (in the case of a not-present violation).

In the case of a canonical violation that was caused intentionally by software, recovery is possible by loading the correct canonical value into RSP. Otherwise, a canonical violation of the address in RSP likely reflects some register corruption in the software.

### Exception Error Code

If the exception is caused by a not-present stack segment or by overflow of the new stack during an inter-privilege-level call, the error code contains a segment selector for the segment that caused the exception. Here, the exception handler can test the present flag in the segment descriptor pointed to by the segment selector to determine the cause of the exception. For a normal limit violation (on a stack segment already in use) the error code is set to 0.

### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. However, when the exception results from attempting to load a not-present stack segment during a task switch, the CS and EIP registers point to the first instruction of the new task.

### Program State Change

A program-state change does not generally accompany a stack-fault exception, because the instruction that generated the fault is not executed. Here, the instruction can be restarted after the exception handler has corrected the stack fault condition.

If a stack fault occurs during a task switch, it occurs after the commit-to-new-task point (see Section 7.3, "Task Switching"). Here, the processor loads all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The stack fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that are more difficult to diagnose. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

## Interrupt 13—General Protection Exception (#GP)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected one of a class of protection violations called “general-protection violations.” The conditions that cause this exception to be generated comprise all the protection violations that do not cause other exceptions to be generated (such as, invalid-TSS, segment-not-present, stack-fault, or page-fault exceptions). The following conditions cause general-protection exceptions to be generated:

- Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
- Transferring execution to a segment that is not executable.
- Writing to a code segment or a read-only data segment.
- Reading from an execute-only code segment.
- Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment.
- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, “Protection.”
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler’s code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, “Privileged Instructions,” for a list of privileged instructions).
- Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT *n* instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.

- If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1. These bits are checked during a write to control registers CR0, CR3, or CR4 that causes a reloading of the page-directory-pointer-table entry.
- Attempting to write a non-zero value into the reserved bits of the MXCSR register.
- Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.

A program or task can be restarted following any general-protection exception. If the exception occurs while attempting to call an interrupt handler, the interrupted program can be restartable, but the interrupt may be lost.

### Exception Error Code

The processor pushes an error code onto the exception handler's stack. If the fault condition was detected while loading a segment descriptor, the error code contains a segment selector to or IDT vector number for the descriptor; otherwise, the error code is 0. The source of the selector in an error code may be any of the following:

- An operand of the instruction.
- A selector from a gate which is the operand of the instruction.
- A selector from a TSS involved in a task switch.
- IDT vector number.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

In general, a program-state change does not accompany a general-protection exception, because the invalid instruction or operation is not executed. An exception handler can be designed to correct all of the conditions that cause general-protection exceptions and restart the program or task without any loss of program continuity.

If a general-protection exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 7.3, "Task Switching"). If it occurs before the commit point, no program state change occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The general-protection exception handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)

### General Protection Exception in 64-bit Mode

The following conditions cause general-protection exceptions in 64-bit mode:

- If the memory address is in a non-canonical form.
- If a segment descriptor memory address is in non-canonical form.
- If the target offset in a destination operand of a call or jmp is in a non-canonical form.
- If a code segment or 64-bit call gate overlaps non-canonical space.
- If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
- If the EFLAGS.NT bit is set in IRET.
- If the stack segment selector of IRET is null when going back to compatibility mode.
- If the stack segment selector of IRET is null going back to CPL3 and 64-bit mode.
- If a null stack segment selector RPL of IRET is not equal to CPL going back to non-CPL3 and 64-bit mode.
- If the proposed new code segment descriptor of IRET has both the D-bit and the L-bit set.

- If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and it has both the D-bit and the L-bit set.
- If the segment descriptor from a 64-bit call gate is in non-canonical space.
- If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
- If the type field of the upper 64 bits of a 64-bit call gate is not 0.
- If an attempt is made to load a null selector in the SS register in compatibility mode.
- If an attempt is made to load null selector in the SS register in CPL3 and 64-bit mode.
- If an attempt is made to load a null selector in the SS register in non-CPL3 and 64-bit mode where RPL is not equal to CPL.
- If an attempt is made to clear CR0.PG while IA-32e mode is enabled.
- If an attempt is made to set a reserved bit in CR3, CR4 or CR8.

## Interrupt 14—Page-Fault Exception (#PF)

**Exception Class**     **Fault.**

### Description

Indicates that, with paging enabled (the PG flag in the CR0 register is set), the processor detected one of the following conditions while using the page-translation mechanism to translate a linear address to a physical address:

- The P (present) flag in a page-directory or page-table entry needed for the address translation is clear, indicating that a page table or the page containing the operand is not present in physical memory.
- The procedure does not have sufficient privilege to access the indicated page (that is, a procedure running in user mode attempts to access a supervisor-mode page). If the SMAP flag is set in CR4, a page fault may also be triggered by code running in supervisor mode that tries to access data at a user-mode address. If the PKE flag is set in CR4, the PKRU register may cause page faults on data accesses to user-mode addresses with certain protection keys.
- Code running in user mode attempts to write to a read-only page. If the WP flag is set in CR0, the page fault will also be triggered by code running in supervisor mode that tries to write to a read-only page.
- An instruction fetch to a linear address that translates to a physical address in a memory page with the execute-disable bit set (for information about the execute-disable bit, see Chapter 4, “Paging”). If the SMEP flag is set in CR4, a page fault will also be triggered by code running in supervisor mode that tries to fetch an instruction from a user-mode address.
- One or more reserved bits in paging-structure entry are set to 1. See description below of RSVD error code flag.
- An enclave access violates one of the specified access-control requirements. See Section 37.3, “Access-control Requirements” and Section 37.19, “Enclave Page Cache Map (EPCM)” in Chapter 37, “Enclave Access Control and Data Structures.” In this case, the exception is called an **SGX-induced page fault**. The processor uses the error code (below) to distinguish SGX-induced page faults from ordinary page faults.

The exception handler can recover from page-not-present conditions and restart the program or task without any loss of program continuity. It can also restart the program or task after a privilege violation, but the problem that caused the privilege violation may be uncorrectable.

See also: Section 4.7, “Page-Fault Exceptions.”

### Exception Error Code

Yes (special format). The processor provides the page-fault handler with two items of information to aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 6-9). The processor establishes the bits in the error code as follows:
  - P flag (bit 0).  
This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
  - W/R (bit 1).  
If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - U/S (bit 2).  
If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - RSVD flag (bit 3).  
This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address.





### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. If the page-fault exception occurred during a task switch, the CS and EIP registers may point to the first instruction of the new task (as described in the following “Program State Change” section).

### Program State Change

A program-state change does not normally accompany a page-fault exception, because the instruction that causes the exception to be generated is not executed. After the page-fault exception handler has corrected the violation (for example, loaded the missing page into memory), execution of the program or task can be resumed.

When a page-fault exception is generated during a task switch, the program-state may change, as follows. During a task switch, a page-fault exception can occur during any of following operations:

- While writing the state of the original task into the TSS of that task.
- While reading the GDT to locate the TSS descriptor of the new task.
- While reading the TSS of the new task.
- While reading segment descriptors associated with segment selectors from the new task.
- While reading the LDT of the new task to verify the segment registers stored in the new TSS.

In the last two cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be called through a task gate.

If a page fault occurs during a task switch, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The page-fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

### Additional Exception-Handling Information

Special care should be taken to ensure that an exception that occurs during an explicit stack switch does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for 16-bit IA-32 processors often use a pair of instructions to change to a new stack, for example:

```
MOV SS, AX
MOV SP, StackTop
```

When executing this code on one of the 32-bit IA-32 processors, it is possible to get a page fault, general-protection fault (#GP), or alignment check fault (#AC) after the segment selector has been loaded into the SS register but before the ESP register has been loaded. At this point, the two parts of the stack pointer (SS and ESP) are inconsistent. The new stack segment is being used with the old stack pointer.

The processor does not use the inconsistent stack pointer if the exception handler switches to a well defined stack (that is, the handler is a task or a more privileged procedure). However, if the exception handler is called at the same privilege level and from the same task, the processor will attempt to use the inconsistent stack pointer.

In systems that handle page-fault, general-protection, or alignment check exceptions within the faulting task (with trap or interrupt gates), software executing at the same privilege level as the exception handler should initialize a new stack by using the LSS instruction rather than a pair of MOV instructions, as described earlier in this note. When the exception handler is running at privilege level 0 (the normal case), the problem is limited to procedures or tasks that run at privilege level 0, typically the kernel of the operating system.

## Interrupt 16—x87 FPU Floating-Point Error (#MF)

**Exception Class**     **Fault.**

### Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, “Control Registers,” for a detailed description of the NE flag.)

### NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

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

Each of these error conditions represents an x87 FPU exception type, and for each of exception type, the x87 FPU provides a flag in the x87 FPU status register and a mask bit in the x87 FPU control register. If the x87 FPU detects a floating-point error and the mask bit for the exception type is set, the x87 FPU handles the exception automatically by generating a predefined (default) response and continuing program execution. The default responses have been designed to provide a reasonable result for most floating-point applications.

If the mask for the exception is clear and the NE flag in register CR0 is set, the x87 FPU does the following:

1. Sets the necessary flag in the FPU status register.
2. Waits until the next “waiting” x87 FPU instruction or WAIT/FWAIT instruction is encountered in the program’s instruction stream.
3. Generates an internal error signal that cause the processor to generate a floating-point exception (#MF).

Prior to executing a waiting x87 FPU instruction or the WAIT/FWAIT instruction, the x87 FPU checks for pending x87 FPU floating-point exceptions (as described in step 2 above). Pending x87 FPU floating-point exceptions are ignored for “non-waiting” x87 FPU instructions, which include the FNINIT, FNCLEX, FNSTSW, FNSTSW AX, FNSTCW, FNSTENV, and FNSAVE instructions. Pending x87 FPU exceptions are also ignored when executing the state management instructions FXSAVE and FXRSTOR.

All of the x87 FPU floating-point error conditions can be recovered from. The x87 FPU floating-point-error exception handler can determine the error condition that caused the exception from the settings of the flags in the x87 FPU status word. See “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on handling x87 FPU floating-point exceptions.

### Exception Error Code

None. The x87 FPU provides its own error information.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the floating-point or WAIT/FWAIT instruction that was about to be executed when the floating-point-error exception was generated. This is not the faulting instruction in which the error condition was detected. The address of the faulting instruction is contained in the x87 FPU instruction pointer

register. See Section 8.1.8, “x87 FPU Instruction and Data (Operand) Pointers” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about information the FPU saves for use in handling floating-point-error exceptions.

### Program State Change

A program-state change generally accompanies an x87 FPU floating-point exception because the handling of the exception is delayed until the next waiting x87 FPU floating-point or WAIT/FWAIT instruction following the faulting instruction. The x87 FPU, however, saves sufficient information about the error condition to allow recovery from the error and re-execution of the faulting instruction if needed.

In situations where non- x87 FPU floating-point instructions depend on the results of an x87 FPU floating-point instruction, a WAIT or FWAIT instruction can be inserted in front of a dependent instruction to force a pending x87 FPU floating-point exception to be handled before the dependent instruction is executed. See “x87 FPU Exception Synchronization” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about synchronization of x87 floating-point-error exceptions.

## Interrupt 17—Alignment Check Exception (#AC)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) accesses (not in code fetches or system segment accesses). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 6-7 lists the alignment requirements various data types recognized by the processor.

**Table 6-7. Alignment Requirements by Data Type**

Data Type	Address Must Be Divisible By
Word	2
Doubleword	4
Single-precision floating-point (32-bits)	4
Double-precision floating-point (64-bits)	8
Double extended-precision floating-point (80-bits)	8
Quadword	8
Double quadword	16
Segment Selector	2
32-bit Far Pointer	2
48-bit Far Pointer	4
32-bit Pointer	4
GDTR, IDTR, LDTR, or Task Register Contents	4
FSTENV/FLDENV Save Area	4 or 2, depending on operand size
FSAVE/FRSTOR Save Area	4 or 2, depending on operand size
Bit String	2 or 4 depending on the operand-size attribute.

Note that the alignment check exception (#AC) is generated only for data types that must be aligned on word, doubleword, and quadword boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:

- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (protected mode or virtual-8086 mode).

Alignment-check exceptions (#AC) are generated only when operating at privilege level 3 (user mode). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate alignment-check exceptions, even when caused by a memory reference made from privilege level 3.

Storing the contents of the GDTR, IDTR, LDTR, or task register in memory while at privilege level 3 can generate an alignment-check exception. Although application programs do not normally store these registers, the fault can be avoided by aligning the information stored on an even word-address.

The FXSAVE/XSAVE and FXRSTOR/XRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the processor implementation (see “FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State” and “FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State” in

Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*; see "XSAVE—Save Processor Extended States" and "XRSTOR—Restore Processor Extended States" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*).

The MOVDQU, MOVUPS, and MOVUPD instructions perform 128-bit unaligned loads or stores. The LDDQU instruction loads 128-bit unaligned data. They do not generate general-protection exceptions (#GP) when operands are not aligned on a 16-byte boundary. If alignment checking is enabled, alignment-check exceptions (#AC) may or may not be generated depending on processor implementation when data addresses are not aligned on an 8-byte boundary.

FSAVE and FRSTOR instructions can generate unaligned references, which can cause alignment-check faults. These instructions are rarely needed by application programs.

### Exception Error Code

Yes. The error code is null; all bits are clear except possibly bit 0 — EXT; see Section 6.13. EXT is set if the #AC is recognized during delivery of an event other than a software interrupt (see "INT n/INTO/INT3/INT1—Call to Interrupt Procedure" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

### Program State Change

A program-state change does not accompany an alignment-check fault, because the instruction is not executed.

## Interrupt 18—Machine-Check Exception (#MC)

**Exception Class**     **Abort.**

### Description

Indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The machine-check exception is model-specific, available on the Pentium and later generations of processors. The implementation of the machine-check exception is different between different processor families, and these implementations may not be compatible with future Intel 64 or IA-32 processors. (Use the CPUID instruction to determine whether this feature is present.)

Bus errors detected by external agents are signaled to the processor on dedicated pins: the BINIT# and MCERR# pins on the Pentium 4, Intel Xeon, and P6 family processors and the BUSCHK# pin on the Pentium processor. When one of these pins is enabled, asserting the pin causes error information to be loaded into machine-check registers and a machine-check exception is generated.

The machine-check exception and machine-check architecture are discussed in detail in Chapter 15, “Machine-Check Architecture.” Also, see the data books for the individual processors for processor-specific hardware information.

### Exception Error Code

None. Error information is provide by machine-check MSRs.

### Saved Instruction Pointer

For the Pentium 4 and Intel Xeon processors, the saved contents of extended machine-check state registers are directly associated with the error that caused the machine-check exception to be generated (see Section 15.3.1.2, “IA32\_MCG\_STATUS MSR,” and Section 15.3.2.6, “IA32\_MCG Extended Machine Check State MSRs”).

For the P6 family processors, if the EIPV flag in the MCG\_STATUS MSR is set, the saved contents of CS and EIP registers are directly associated with the error that caused the machine-check exception to be generated; if the flag is clear, the saved instruction pointer may not be associated with the error (see Section 15.3.1.2, “IA32\_MCG\_STATUS MSR”).

For the Pentium processor, contents of the CS and EIP registers may not be associated with the error.

### Program State Change

The machine-check mechanism is enabled by setting the MCE flag in control register CR4.

For the Pentium 4, Intel Xeon, P6 family, and Pentium processors, a program-state change always accompanies a machine-check exception, and an abort class exception is generated. For abort exceptions, information about the exception can be collected from the machine-check MSRs, but the program cannot generally be restarted.

If the machine-check mechanism is not enabled (the MCE flag in control register CR4 is clear), a machine-check exception causes the processor to enter the shutdown state.

## Interrupt 19—SIMD Floating-Point Exception (#XM)

**Exception Class**     **Fault.**

### Description

Indicates the processor has detected an SSE/SSE2/SSE3 SIMD floating-point exception. The appropriate status flag in the MXCSR register must be set and the particular exception unmasked for this interrupt to be generated.

There are six classes of numeric exception conditions that can occur while executing an SSE/ SSE2/SSE3 SIMD floating-point instruction:

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

The invalid operation, divide-by-zero, and denormal-operand exceptions are pre-computation exceptions; that is, they are detected before any arithmetic operation occurs. The numeric underflow, numeric overflow, and inexact result exceptions are post-computational exceptions.

See “SIMD Floating-Point Exceptions” in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information about the SIMD floating-point exception classes.

When a SIMD floating-point exception occurs, the processor does either of the following things:

- It handles the exception automatically by producing the most reasonable result and allowing program execution to continue undisturbed. This is the response to masked exceptions.
- It generates a SIMD floating-point exception, which in turn invokes a software exception handler. This is the response to unmasked exceptions.

Each of the six SIMD floating-point exception conditions has a corresponding flag bit and mask bit in the MXCSR register. If an exception is masked (the corresponding mask bit in the MXCSR register is set), the processor takes an appropriate automatic default action and continues with the computation. If the exception is unmasked (the corresponding mask bit is clear) and the operating system supports SIMD floating-point exceptions (the OSXM-MEXCPT flag in control register CR4 is set), a software exception handler is invoked through a SIMD floating-point exception. If the exception is unmasked and the OSXMMEXCPT bit is clear (indicating that the operating system does not support unmasked SIMD floating-point exceptions), an invalid opcode exception (#UD) is signaled instead of a SIMD floating-point exception.

Note that because SIMD floating-point exceptions are precise and occur immediately, the situation does not arise where an x87 FPU instruction, a WAIT/FWAIT instruction, or another SSE/SSE2/SSE3 instruction will catch a pending unmasked SIMD floating-point exception.

In situations where a SIMD floating-point exception occurred while the SIMD floating-point exceptions were masked (causing the corresponding exception flag to be set) and the SIMD floating-point exception was subsequently unmasked, then no exception is generated when the exception is unmasked.

When SSE/SSE2/SSE3 SIMD floating-point instructions operate on packed operands (made up of two or four sub-operands), multiple SIMD floating-point exception conditions may be detected. If no more than one exception condition is detected for one or more sets of sub-operands, the exception flags are set for each exception condition detected. For example, an invalid exception detected for one sub-operand will not prevent the reporting of a divide-by-zero exception for another sub-operand. However, when two or more exceptions conditions are generated for one sub-operand, only one exception condition is reported, according to the precedences shown in Table 6-8. This exception precedence sometimes results in the higher priority exception condition being reported and the lower priority exception conditions being ignored.

**Table 6-8. SIMD Floating-Point Exceptions Priority**

Priority	Description
1 (Highest)	Invalid operation exception due to SNaN operand (or any NaN operand for maximum, minimum, or certain compare and convert operations).
2	QNaN operand <sup>1</sup> .
3	Any other invalid operation exception not mentioned above or a divide-by-zero exception <sup>2</sup> .
4	Denormal operand exception <sup>2</sup> .
5	Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception <sup>2</sup> .
6 (Lowest)	Inexact result exception.

**NOTES:**

1. Though a QNaN this is not an exception, the handling of a QNaN operand has precedence over lower priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a divide-by-zero- exception.
2. If masked, then instruction execution continues, and a lower priority exception can occur as well.

**Exception Error Code**

None.

**Saved Instruction Pointer**

The saved contents of CS and EIP registers point to the SSE/SSE2/SSE3 instruction that was executed when the SIMD floating-point exception was generated. This is the faulting instruction in which the error condition was detected.

**Program State Change**

A program-state change does not accompany a SIMD floating-point exception because the handling of the exception is immediate unless the particular exception is masked. The available state information is often sufficient to allow recovery from the error and re-execution of the faulting instruction if needed.



## Interrupt 20—Virtualization Exception (#VE)

**Exception Class**     **Fault.**

### Description

Indicates that the processor detected an EPT violation in VMX non-root operation. Not all EPT violations cause virtualization exceptions. See Section 25.5.6.2 for details.

The exception handler can recover from EPT violations and restart the program or task without any loss of program continuity. In some cases, however, the problem that caused the EPT violation may be uncorrectable.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception.

### Program State Change

A program-state change does not normally accompany a virtualization exception, because the instruction that causes the exception to be generated is not executed. After the virtualization exception handler has corrected the violation (for example, by executing the EPTP-switching VM function), execution of the program or task can be resumed.

### Additional Exception-Handling Information

The processor saves information about virtualization exceptions in the virtualization-exception information area. See Section 25.5.6.2 for details.

## Interrupts 32 to 255—User Defined Interrupts

**Exception Class**     **Not applicable.**

### Description

Indicates that the processor did one of the following things:

- Executed an INT  $n$  instruction where the instruction operand is one of the vector numbers from 32 through 255.
- Responded to an interrupt request at the INTR pin or from the local APIC when the interrupt vector number associated with the request is from 32 through 255.

### Exception Error Code

Not applicable.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that follows the INT  $n$  instruction or instruction following the instruction on which the INTR signal occurred.

### Program State Change

A program-state change does not accompany interrupts generated by the INT  $n$  instruction or the INTR signal. The INT  $n$  instruction generates the interrupt within the instruction stream. When the processor receives an INTR signal, it commits all state changes for all previous instructions before it responds to the interrupt; so, program execution can resume upon returning from the interrupt handler.



## 10. Updates to Chapter 7, Volume 3A

Change bars show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----

Changes to this chapter: Updates to Section 7.3 "Task Switching" and Section 7.7 "Task Management in 64-bit Mode".

This chapter describes the IA-32 architecture's task management facilities. These facilities are only available when the processor is running in protected mode.

This chapter focuses on 32-bit tasks and the 32-bit TSS structure. For information on 16-bit tasks and the 16-bit TSS structure, see Section 7.6, "16-Bit Task-State Segment (TSS)." For information specific to task management in 64-bit mode, see Section 7.7, "Task Management in 64-bit Mode."

## 7.1 TASK MANAGEMENT OVERVIEW

A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.

The IA-32 architecture provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another. When operating in protected mode, all processor execution takes place from within a task. Even simple systems must define at least one task. More complex systems can use the processor's task management facilities to support multitasking applications.

### 7.1.1 Task Structure

A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments (see Figure 7-1). If an operating system or executive uses the processor's privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level.

The TSS specifies the segments that make up the task execution space and provides a storage place for task state information. In multitasking systems, the TSS also provides a mechanism for linking tasks.

A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for the TSS are loaded into the task register (see Section 2.4.4, "Task Register (TR)").

If paging is implemented for the task, the base address of the page directory used by the task is loaded into control register CR3.

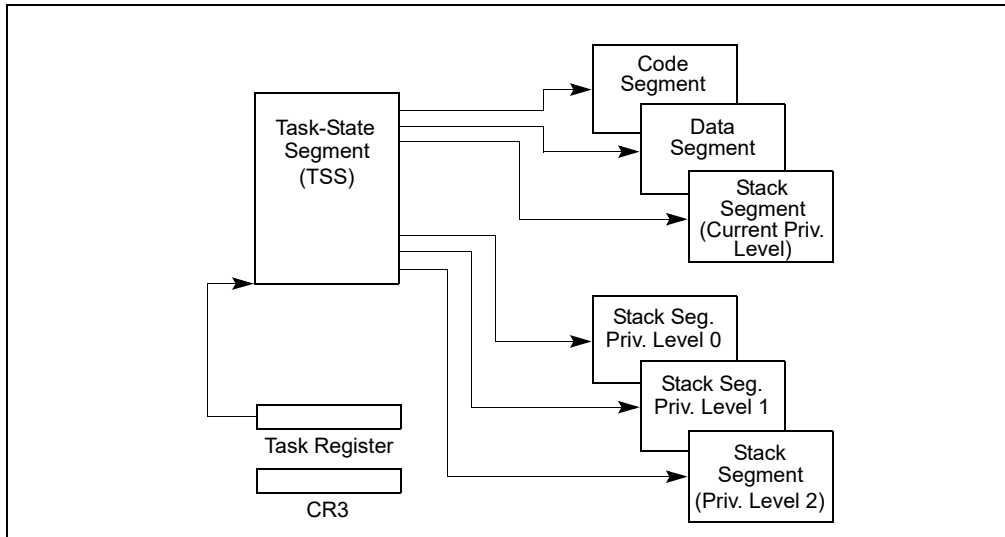


Figure 7-1. Structure of a Task

## 7.1.2 Task State

The following items define the state of the currently executing task:

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.
- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).
- Link to previously executed task (contained in the TSS).

Prior to dispatching a task, all of these items are contained in the task's TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

## 7.1.3 Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

- A explicit call to a task with the CALL instruction.
- A explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods for dispatching a task identify the task to be dispatched with a segment selector that points to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task

to handle an interrupt or exception, the IDT entry for the interrupt or exception must contain a task gate that holds the selector for the interrupt- or exception-handler TSS.

When a task is dispatched for execution, a task switch occurs between the currently running task and the dispatched task. During a task switch, the execution environment of the currently executing task (called the task's state or context) is saved in its TSS and execution of the task is suspended. The context for the dispatched task is then loaded into the processor and execution of that task begins with the instruction pointed to by the newly loaded EIP register. If the task has not been run since the system was last initialized, the EIP will point to the first instruction of the task's code; otherwise, it will point to the next instruction after the last instruction that the task executed when it was last active.

If the currently executing task (the calling task) called the task being dispatched (the called task), the TSS segment selector for the calling task is stored in the TSS of the called task to provide a link back to the calling task.

For all IA-32 processors, tasks are not recursive. A task cannot call or jump to itself.

Interrupts and exceptions can be handled with a task switch to a handler task. Here, the processor performs a task switch to handle the interrupt or exception and automatically switches back to the interrupted task upon returning from the interrupt-handler task or exception-handler task. This mechanism can also handle interrupts that occur during interrupt tasks.

As part of a task switch, the processor can also switch to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. The page-directory base register (CR3) also is reloaded on a task switch, allowing each task to have its own set of page tables. These protection facilities help isolate tasks and prevent them from interfering with one another.

If protection mechanisms are not used, the processor provides no protection between tasks. This is true even with operating systems that use multiple privilege levels for protection. A task running at privilege level 3 that uses the same LDT and page tables as other privilege-level-3 tasks can access code and corrupt data and the stack of other tasks.

Use of task management facilities for handling multitasking applications is optional. Multitasking can be handled in software, with each software defined task executed in the context of a single IA-32 architecture task.

## 7.2 TASK MANAGEMENT DATA STRUCTURES

The processor defines five data structures for handling task-related activities:

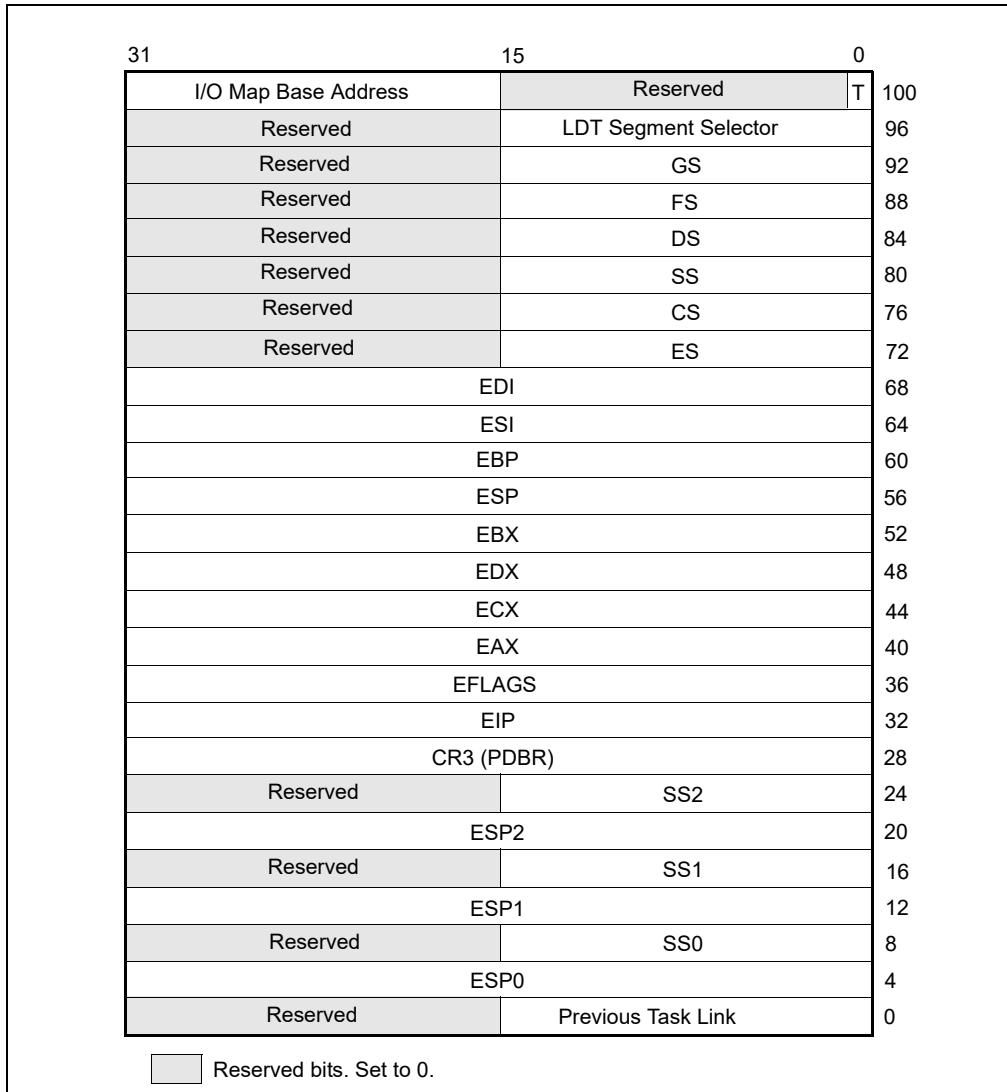
- Task-state segment (TSS).
- Task-gate descriptor.
- TSS descriptor.
- Task register.
- NT flag in the EFLAGS register.

When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

### 7.2.1 Task-State Segment (TSS)

The processor state information needed to restore a task is saved in a system segment called the task-state segment (TSS). Figure 7-2 shows the format of a TSS for tasks designed for 32-bit CPUs. The fields of a TSS are divided into two main categories: dynamic fields and static fields.

For information about 16-bit Intel 286 processor task structures, see Section 7.6, "16-Bit Task-State Segment (TSS)." For information about 64-bit mode task structures, see Section 7.7, "Task Management in 64-bit Mode."



**Figure 7-2. 32-Bit Task-State Segment (TSS)**

The processor updates dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

- **General-purpose register fields** — State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.
- **Segment selector fields** — Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.
- **EFLAGS register field** — State of the EFAGS register prior to the task switch.
- **EIP (instruction pointer) field** — State of the EIP register prior to the task switch.
- **Previous task link field** — Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field (which is sometimes called the back link field) permits a task switch back to the previous task by using the IRET instruction.

The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

- **LDT segment selector field** — Contains the segment selector for the task's LDT.



- **CR3 control register field** — Contains the base physical address of the page directory to be used by the task. Control register CR3 is also known as the page-directory base register (PDBR).
- **Privilege level-0, -1, and -2 stack pointer fields** — These stack pointers consist of a logical address made up of the segment selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack (ESP0, ESP1, and ESP2). Note that the values in these fields are static for a particular task; whereas, the SS and ESP values will change if stack switching occurs within the task.
- **T (debug trap) flag (byte 100, bit 0)** — When set, the T flag causes the processor to raise a debug exception when a task switch to this task occurs (see Section 17.3.1.5, “Task-Switch Exception Condition”).
- **I/O map base address field** — Contains a 16-bit offset from the base of the TSS to the I/O permission bit map and interrupt redirection bitmap. When present, these maps are stored in the TSS at higher addresses. The I/O map base address points to the beginning of the I/O permission bit map and the end of the interrupt redirection bit map. See Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the I/O permission bit map. See Section 20.3, “Interrupt and Exception Handling in Virtual-8086 Mode,” for a detailed description of the interrupt redirection bit map.

If paging is used:

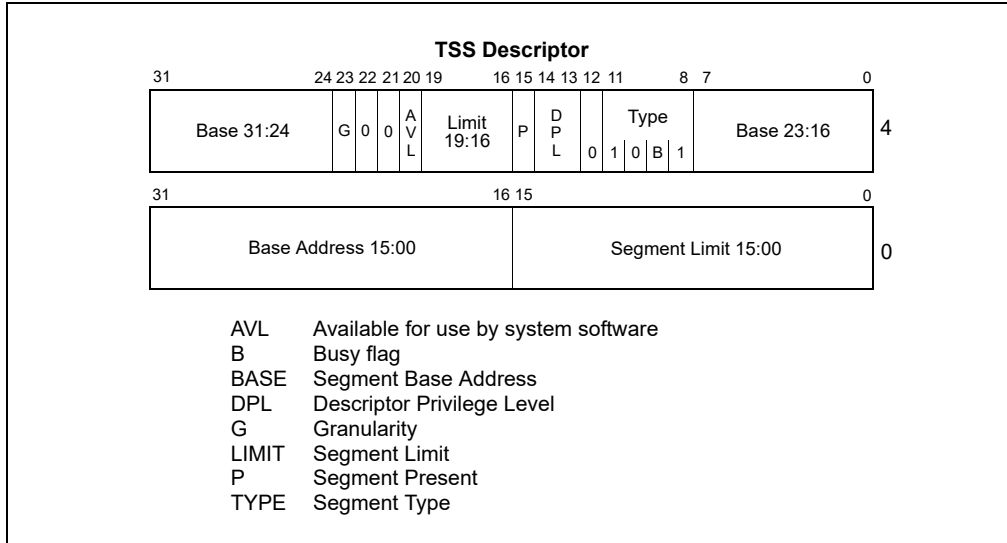
- Avoid placing a page boundary in the part of the TSS that the processor reads during a task switch (the first 104 bytes). The processor may not correctly perform address translations if a boundary occurs in this area. During a task switch, the processor reads and writes into the first 104 bytes of each TSS (using contiguous physical addresses beginning with the physical address of the first byte of the TSS). So, after TSS access begins, if part of the 104 bytes is not physically contiguous, the processor will access incorrect information without generating a page-fault exception.
- Pages corresponding to the previous task’s TSS, the current task’s TSS, and the descriptor table entries for each all should be marked as read/write.
- Task switches are carried out faster if the pages containing these structures are present in memory before the task switch is initiated.

## 7.2.2 TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 7-3 shows the format of a TSS descriptor. TSS descriptors may only be placed in the GDT; they cannot be placed in an LDT or the IDT.

An attempt to access a TSS using a segment selector with its TI flag set (which indicates the current LDT) causes a general-protection exception (#GP) to be generated during CALLs and JMPs; it causes an invalid TSS exception (#TS) during IRETs. A general-protection exception is also generated if an attempt is made to load a segment selector for a TSS into a segment register.

The busy flag (B) in the type field indicates whether the task is busy. A busy task is currently running or suspended. A type field with a value of 1001B indicates an inactive task; a value of 1011B indicates a busy task. Tasks are not recursive. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.



**Figure 7-3. TSS Descriptor**

The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data-segment descriptors (see Section 3.4.5, “Segment Descriptors”). When the G flag is 0 in a TSS descriptor for a 32-bit TSS, the limit field must have a value equal to or greater than 67H, one byte less than the minimum size of a TSS. Attempting to switch to a task whose TSS descriptor has a limit less than 67H generates an invalid-TSS exception (#TS). A larger limit is required if an I/O permission bit map is included or if the operating system stores additional data. The processor does not check for a limit greater than 67H on a task switch; however, it does check when accessing the I/O permission bit map or interrupt redirection bit map.

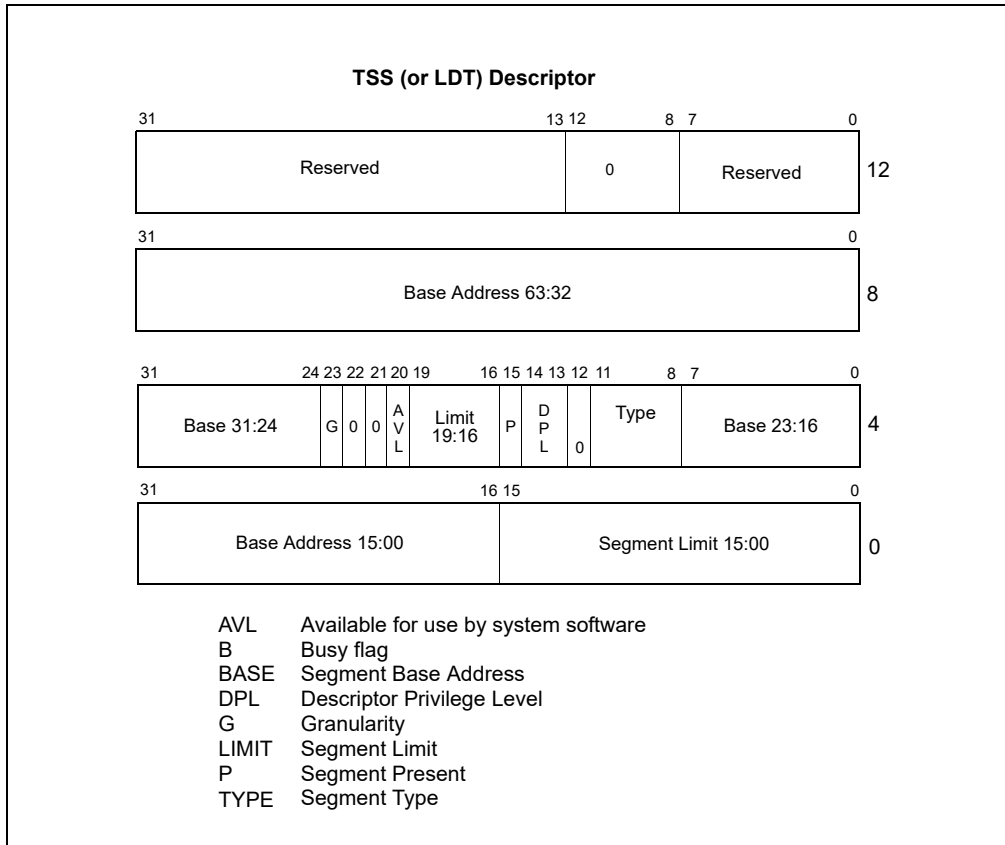
Any program or procedure with access to a TSS descriptor (that is, whose CPL is numerically equal to or less than the DPL of the TSS descriptor) can dispatch the task with a call or a jump.

In most systems, the DPLs of TSS descriptors are set to values less than 3, so that only privileged software can perform task switching. However, in multitasking applications, DPLs for some TSS descriptors may be set to 3 to allow task switching at the application (or user) privilege level.

### 7.2.3 TSS Descriptor in 64-bit mode

In 64-bit mode, task switching is not supported, but TSS descriptors still exist. The format of a 64-bit TSS is described in Section 7.7.

In 64-bit mode, the TSS descriptor is expanded to 16 bytes (see Figure 7-4). This expansion also applies to an LDT descriptor in 64-bit mode. Table 3-2 provides the encoding information for the segment type field.



**Figure 7-4. Format of TSS and LDT Descriptors in 64-bit Mode**

## 7.2.4 Task Register

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address (64 bits in IA-32e mode), 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-6). This information is copied from the TSS descriptor in the GDT for the current task. Figure 7-5 shows the path the processor uses to access the TSS (using the information in the task register).

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

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

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level in order to identify the currently running task. However, it is normally used only by operating system software. (If CR4.UMIP = 1, STR can be executed only when CPL = 0.)

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

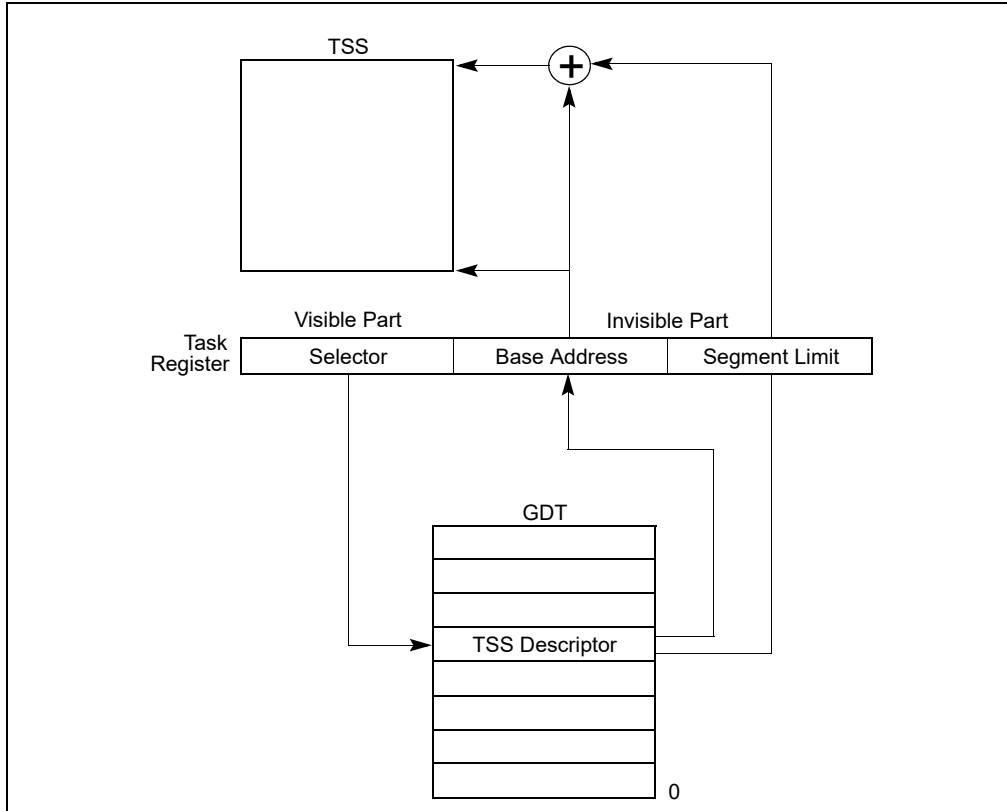


Figure 7-5. Task Register

### 7.2.5 Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task (see Figure 7-6). It can be placed in the GDT, an LDT, or the IDT. The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. The RPL in this segment selector is not used.

The DPL of a task-gate descriptor controls access to the TSS descriptor during a task switch. When a program or procedure makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. Note that when a task gate is used, the DPL of the destination TSS descriptor is not used.

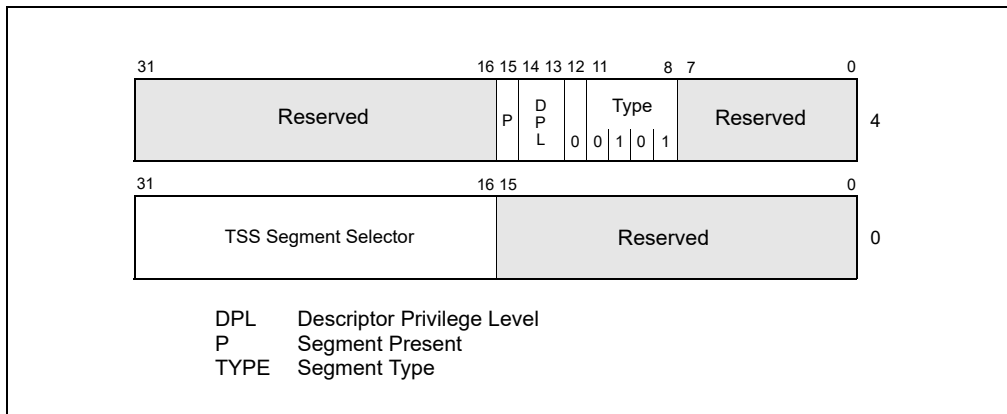


Figure 7-6. Task-Gate Descriptor

A task can be accessed either through a task-gate descriptor or a TSS descriptor. Both of these structures satisfy the following needs:

- **Need for a task to have only one busy flag** — Because the busy flag for a task is stored in the TSS descriptor, each task should have only one TSS descriptor. There may, however, be several task gates that reference the same TSS descriptor.
- **Need to provide selective access to tasks** — Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor's DPL. A program or procedure that does not have sufficient privilege to access the TSS descriptor for a task in the GDT (which usually has a DPL of 0) may be allowed access to the task through a task gate with a higher DPL. Task gates give the operating system greater latitude for limiting access to specific tasks.
- **Need for an interrupt or exception to be handled by an independent task** — Task gates may also reside in the IDT, which allows interrupts and exceptions to be handled by handler tasks. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

Figure 7-7 illustrates how a task gate in an LDT, a task gate in the GDT, and a task gate in the IDT can all point to the same task.

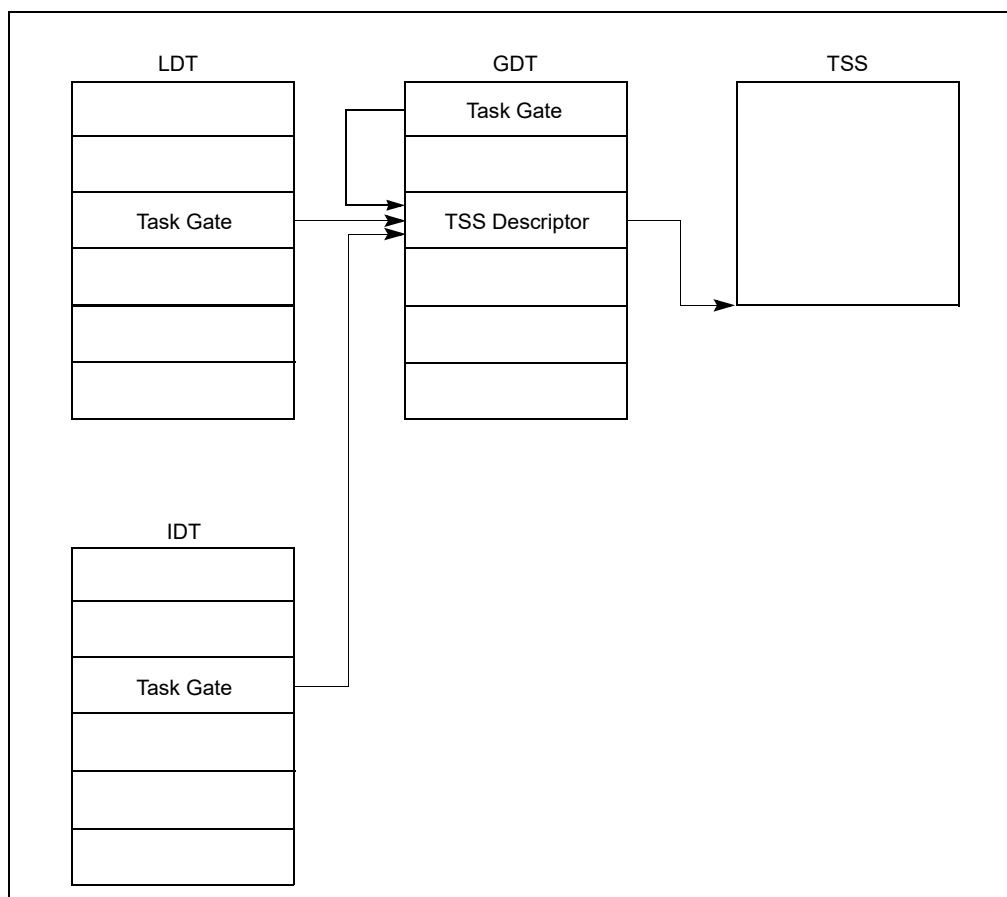


Figure 7-7. Task Gates Referencing the Same Task

## 7.3 TASK SWITCHING

The processor transfers execution to another task in one of four cases:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.

- An interrupt or exception vector points to a task-gate descriptor in the IDT.
- The current task executes an IRET when the NT flag in the EFLAGS register is set.

JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).
2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for those identified in the next sentence), and the IRET and INT1 instructions are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT  $n$ , INT3, and INTO instructions, the DPL is checked and a general-protection exception (#GP) results if it is less than the CPL.<sup>1</sup>
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).
5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt: the busy (B) flag is left set. (See Table 7-2.)
7. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
8. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).
9. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor will set the NT flag in the EFLAGS loaded from the new task. If initiated with an IRET instruction or JMP instruction, the NT flag will reflect the state of NT in the EFLAGS loaded from the new task (see Table 7-2).
10. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
11. Loads the task register with the segment selector and descriptor for the new task's TSS.
12. The TSS state is loaded into the processor. This includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment selectors. A fault during the load of this state may corrupt architectural state. (If paging is not enabled, a PDBR value is read from the new task's TSS, but it is not loaded into CR3.)
13. The descriptors associated with the segment selectors are loaded and qualified. Any errors associated with this loading and qualification occur in the context of the new task and may corrupt architectural state.

### NOTES

If all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 11, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

1. The INT1 has opcode F1; the INT  $n$  instruction with  $n=1$  has opcode CD 01.

If an unrecoverable error occurs in step 12, architectural state may be corrupted, but an attempt will be made to handle the error in the prior execution environment. If an unrecoverable error occurs after the commit point (in step 13), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task.

If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 6, “Interrupt 10—Invalid TSS Exception (#TS),” for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

14. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

Table 7-1 shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other IA-32 processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

**Table 7-1. Exception Conditions Checked During a Task Switch**

Condition Checked	Exception <sup>1</sup>	Error Code Reference <sup>2</sup>
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP	New Task's TSS
P bit is set in TSS descriptor.	#TS (for IRET)	
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS (for IRET)	New Task's TSS
	#TS	New Task's TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid <sup>3</sup> .	#TS	New Task's LDT
If code segment is non-conforming, its DPL should equal its RPL.	#TS	New Code Segment
If code segment is conforming, its DPL should be less than or equal to its RPL.	#TS	New Code Segment
SS segment selector is valid <sup>2</sup> .	#TS	New Stack Segment
P bit is set in stack segment descriptor.	#SS	New Stack Segment
Stack segment DPL should equal CPL.	#TS	New stack segment
P bit is set in new task's LDT descriptor.	#TS	New Task's LDT
CS segment selector is valid <sup>3</sup> .	#TS	New Code Segment
P bit is set in code segment descriptor.	#NP	New Code Segment
Stack segment DPL should equal its RPL.	#TS	New Stack Segment
DS, ES, FS, and GS segment selectors are valid <sup>3</sup> .	#TS	New Data Segment

**Table 7-1. Exception Conditions Checked During a Task Switch (Contd.)**

Condition Checked	Exception <sup>1</sup>	Error Code Reference <sup>2</sup>
DS, ES, FS, and GS segments are readable.	#TS	New Data Segment
P bits are set in descriptors of DS, ES, FS, and GS segments.	#NP	New Data Segment
DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).	#TS	New Data Segment

**NOTES:**

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SS is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).

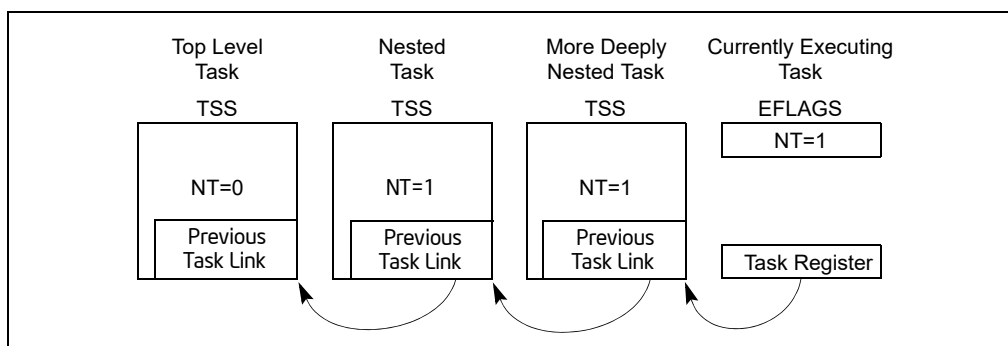
The TS (task switched) flag in the control register CR0 is set every time a task switch occurs. System software uses the TS flag to coordinate the actions of floating-point unit when generating floating-point exceptions with the rest of the processor. The TS flag indicates that the context of the floating-point unit may be different from that of the current task. See Section 2.5, "Control Registers", for a detailed description of the function and use of the TS flag.

## 7.4 TASK LINKING

The previous task link field of the TSS (sometimes called the "backlink") and the NT flag in the EFLAGS register are used to return execution to the previous task. EFLAGS.NT = 1 indicates that the currently executing task is nested within the execution of another task.

When a CALL instruction, an interrupt, or an exception causes a task switch: the processor copies the segment selector for the current TSS to the previous task link field of the TSS for the new task; it then sets EFLAGS.NT = 1. If software uses an IRET instruction to suspend the new task, the processor checks for EFLAGS.NT = 1; it then uses the value in the previous task link field to return to the previous task. See Figures 7-8.

When a JMP instruction causes a task switch, the new task is not nested. The previous task link field is not used and EFLAGS.NT = 0. Use a JMP instruction to dispatch a new task when nesting is not desired.



**Figure 7-8. Nested Tasks**

Table 7-2 shows the busy flag (in the TSS segment descriptor), the NT flag, the previous task link field, and TS flag (in control register CR0) during a task switch.

The NT flag may be modified by software executing at any privilege level. It is possible for a program to set the NT flag and execute an IRET instruction. This might randomly invoke the task specified in the previous link field of the current task's TSS. To keep such spurious task switches from succeeding, the operating system should initialize the



previous task link field in every TSS that it creates to 0.

**Table 7-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag**

Flag or Field	Effect of JMP instruction	Effect of CALL Instruction or Interrupt	Effect of IRET Instruction
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	Set to value from TSS of new task.	Flag is set.	Set to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS.	No change.
Previous task link field of old task.	No change.	No change.	No change.
TS flag in control register CR0.	Flag is set.	Flag is set.	Flag is set.

### 7.4.1 Use of Busy Flag To Prevent Recursive Task Switching

A TSS allows only one context to be saved for a task; therefore, once a task is called (dispatched), a recursive (or re-entrant) call to the task would cause the current state of the task to be lost. The busy flag in the TSS segment descriptor is provided to prevent re-entrant task switching and a subsequent loss of task state information. The processor manages the busy flag as follows:

1. When dispatching a task, the processor sets the busy flag of the new task.
2. If during a task switch, the current task is placed in a nested chain (the task switch is being generated by a CALL instruction, an interrupt, or an exception), the busy flag for the current task remains set.
3. When switching to the new task (initiated by a CALL instruction, interrupt, or exception), the processor generates a general-protection exception (#GP) if the busy flag of the new task is already set. If the task switch is initiated with an IRET instruction, the exception is not raised because the processor expects the busy flag to be set.
4. When a task is terminated by a jump to a new task (initiated with a JMP instruction in the task code) or by an IRET instruction in the task code, the processor clears the busy flag, returning the task to the "not busy" state.

The processor prevents recursive task switching by preventing a task from switching to itself or to any task in a nested chain of tasks. The chain of nested suspended tasks may grow to any length, due to multiple calls, interrupts, or exceptions. The busy flag prevents a task from being invoked if it is in this chain.

The busy flag may be used in multiprocessor configurations, because the processor follows a LOCK protocol (on the bus or in the cache) when it sets or clears the busy flag. This lock keeps two processors from invoking the same task at the same time. See Section 8.1.2.1, "Automatic Locking," for more information about setting the busy flag in a multiprocessor applications.

### 7.4.2 Modifying Task Linkages

In a uniprocessor system, in situations where it is necessary to remove a task from a chain of linked tasks, use the following procedure to remove the task:

1. Disable interrupts.
2. Change the previous task link field in the TSS of the pre-empting task (the task that suspended the task to be removed). It is assumed that the pre-empting task is the next task (newer task) in the chain from the task to be removed. Change the previous task link field to point to the TSS of the next oldest task in the chain or to an even older task in the chain.

3. Clear the busy (B) flag in the TSS segment descriptor for the task being removed from the chain. If more than one task is being removed from the chain, the busy flag for each task being removed must be cleared.
4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to insure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

## 7.5 TASK ADDRESS SPACE

The address space for a task consists of the segments that the task can access. These segments include the code, data, stack, and system segments referenced in the TSS and any other segments accessed by the task code. The segments are mapped into the processor's linear address space, which is in turn mapped into the processor's physical address space (either directly or through paging).

The LDT segment field in the TSS can be used to give each task its own LDT. Giving a task its own LDT allows the task address space to be isolated from other tasks by placing the segment descriptors for all the segments associated with the task in the task's LDT.

It also is possible for several tasks to use the same LDT. This is a memory-efficient way to allow specific tasks to communicate with or control each other, without dropping the protection barriers for the entire system.

Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

If paging is enabled, the CR3 register (PDBR) field in the TSS allows each task to have its own set of page tables for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

### 7.5.1 Mapping Tasks to the Linear and Physical Address Spaces

Tasks can be mapped to the linear address space and physical address space in one of two ways:

- **One linear-to-physical address space mapping is shared among all tasks.** — When paging is not enabled, this is the only choice. Without paging, all linear addresses map to the same physical addresses. When paging is enabled, this form of linear-to-physical address space mapping is obtained by using one page directory for all tasks. The linear address space may exceed the available physical space if demand-paged virtual memory is supported.
- **Each task has its own linear address space that is mapped to the physical address space.** — This form of mapping is accomplished by using a different page directory for each task. Because the PDBR (control register CR3) is loaded on task switches, each task may have a different page directory.

The linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share physical addresses.

With either method of mapping task linear address spaces, the TSSs for all tasks must lie in a shared area of the physical space, which is accessible to all tasks. This mapping is required so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear address space mapped by the GDT also should be mapped to a shared area of the physical space; otherwise, the purpose of the GDT is defeated. Figure 7-9 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

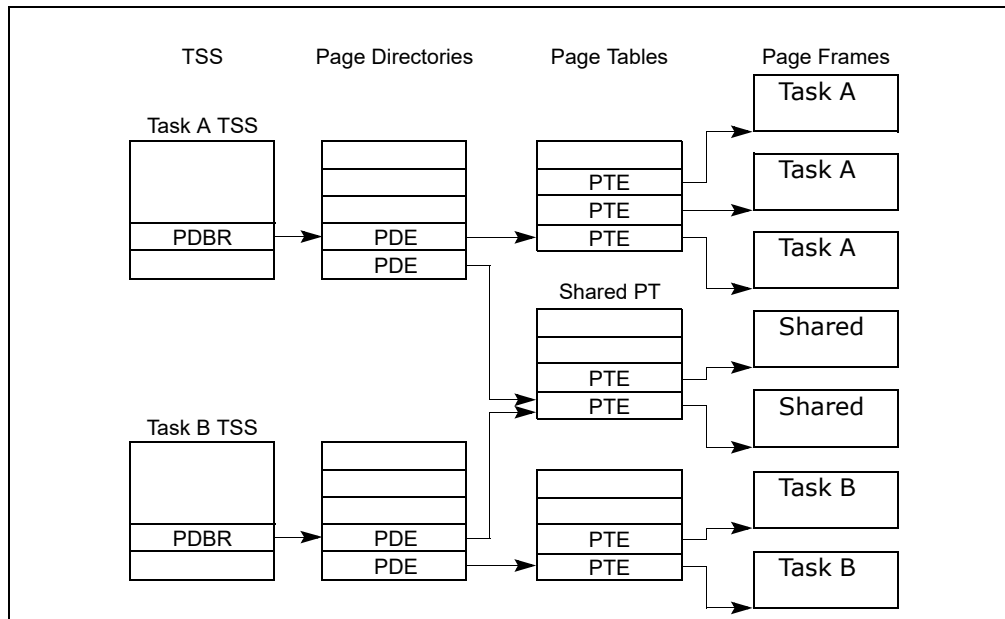


Figure 7-9. Overlapping Linear-to-Physical Mappings

## 7.5.2 Task Logical Address Space

To allow the sharing of data among tasks, use the following techniques to create shared logical-to-physical address-space mappings for data segments:

- **Through the segment descriptors in the GDT** — All tasks must have access to the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear-address space that are mapped into an area of the physical-address space common to all tasks, then all tasks can share the data and code in those segments.
- **Through a shared LDT** — Two or more tasks can use the same LDT if the LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments that are mapped to a common area of the physical address space, the data and code in those segments can be shared among the tasks that share the LDT. This method of sharing is more selective than sharing through the GDT, because the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared segments.
- **Through segment descriptors in distinct LDTs that are mapped to common addresses in linear address space** — If this common area of the linear address space is mapped to the same area of the physical address space for each task, these segment descriptors permit the tasks to share segments. Such segment descriptors are commonly called aliases. This method of sharing is even more selective than those listed above, because, other segment descriptors in the LDTs may point to independent linear addresses which are not shared.

## 7.6 16-BIT TASK-STATE SEGMENT (TSS)

The 32-bit IA-32 processors also recognize a 16-bit TSS format like the one used in Intel 286 processors (see Figure 7-10). This format is supported for compatibility with software written to run on earlier IA-32 processors.

The following information is important to know about the 16-bit TSS.

- Do not use a 16-bit TSS to implement a virtual-8086 task.
- The valid segment limit for a 16-bit TSS is 2CH.

- The 16-bit TSS does not contain a field for the base address of the page directory, which is loaded into control register CR3. A separate set of page tables for each task is not supported for 16-bit tasks. If a 16-bit task is dispatched, the page-table structure for the previous task is used.
- The I/O base address is not included in the 16-bit TSS. None of the functions of the I/O map are supported.
- When task state is saved in a 16-bit TSS, the upper 16 bits of the EFLAGS register and the EIP register are lost.
- When the general-purpose registers are loaded or saved from a 16-bit TSS, the upper 16 bits of the registers are modified and not maintained.

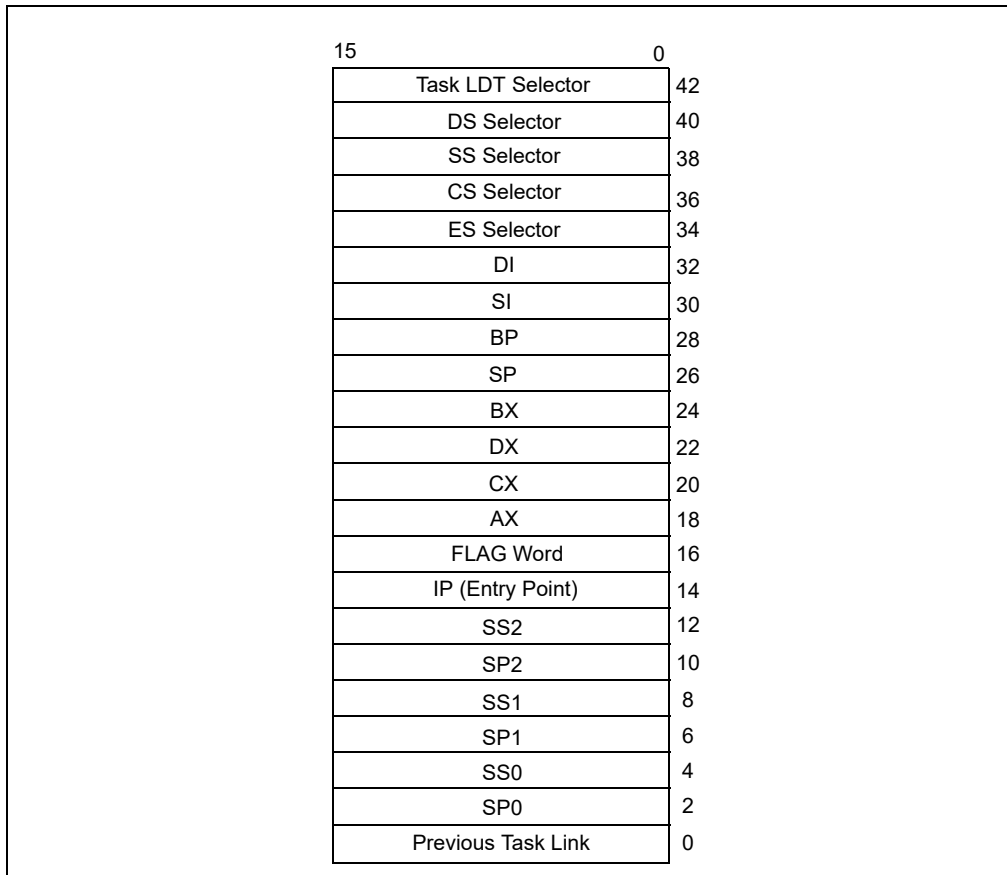


Figure 7-10. 16-Bit TSS Format

## 7.7 TASK MANAGEMENT IN 64-BIT MODE

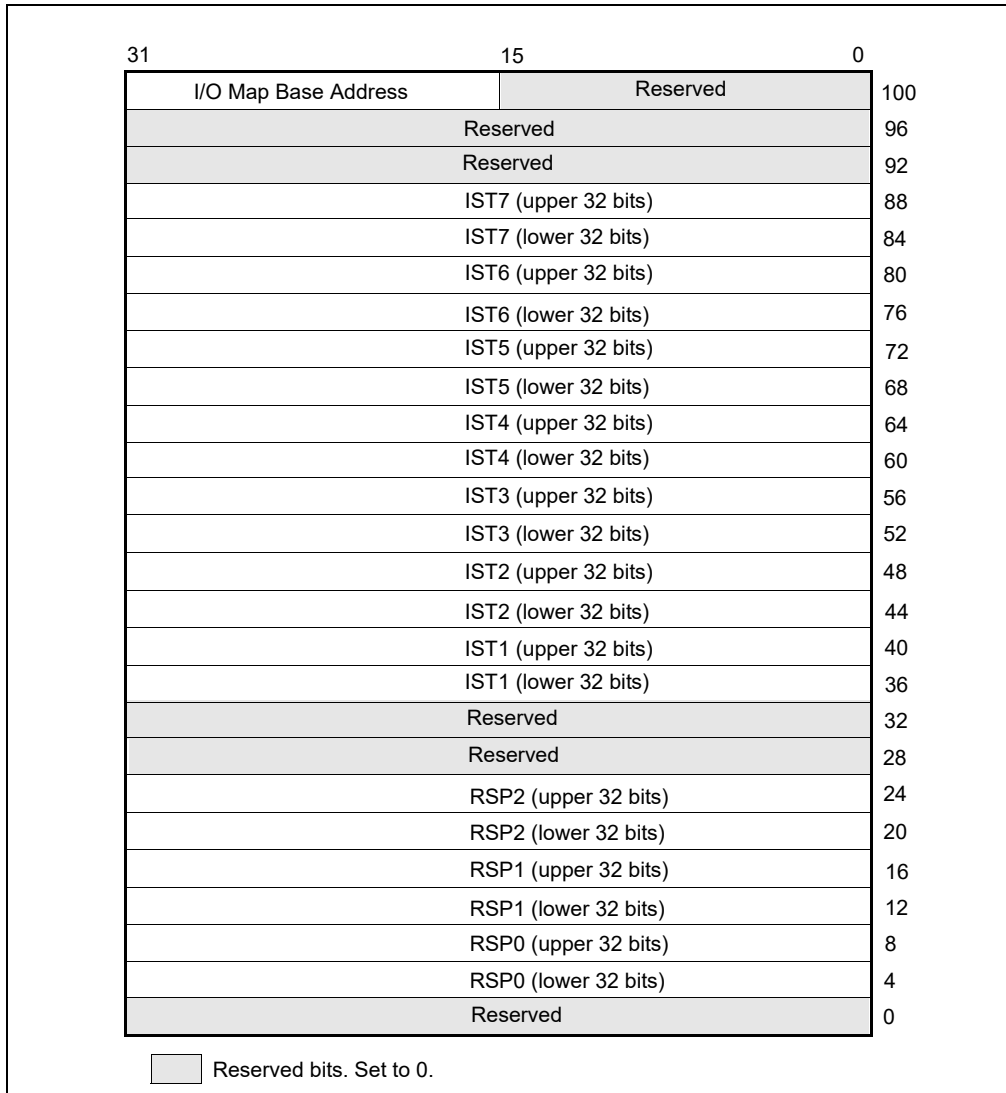
In 64-bit mode, task structure and task state are similar to those in protected mode. However, the task switching mechanism available in protected mode is not supported in 64-bit mode. Task management and switching must be performed by software. The processor issues a general-protection exception (#GP) if the following is attempted in 64-bit mode:

- Control transfer to a TSS or a task gate using JMP, CALL, INT *n*, INT3, INTO, INT1, or interrupt.
- An IRET with EFLAGS.NT (nested task) set to 1.

Although hardware task-switching is not supported in 64-bit mode, a 64-bit task state segment (TSS) must exist. Figure 7-11 shows the format of a 64-bit TSS. The TSS holds information important to 64-bit mode and that is not directly related to the task-switch mechanism. This information includes:

- **RSPn** — The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0-2.
- **ISTn** — The full 64-bit canonical forms of the interrupt stack table (IST) pointers.
- **I/O map base address** — The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.

The operating system must create at least one 64-bit TSS after activating IA-32e mode. It must execute the LTR instruction (in 64-bit mode) to load the TR register with a pointer to the 64-bit TSS responsible for both 64-bit-mode programs and compatibility-mode programs.



**Figure 7-11. 64-Bit TSS Format**



## 11. Updates to Chapter 17, Volume 3B

Change bars show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter: Updates to Section 17.1 "Overview of Debug Support Facilities", Section 17.2.5 "Breakpoint Field Recognition", Section 17.3.1 "Debug Exception (#DB)—Interrupt Vector 1", Section 17.3.1.1 "Instruction-Breakpoint Exception Condition", Section 17.3.1.2 "Data Memory and I/O Breakpoint Exception Conditions", Section 17.3.1.4 "Single-Step Exception Condition", and Section 17.3.2 "Breakpoint Exception (#BP)—Interrupt Vector 3".

# CHAPTER 17

## DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FEATURES

---

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

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.
- Time stamp counter is described in Section 17.17, “Time-Stamp Counter”.
- Features which allow monitoring of shared platform resources such as the L3 cache are described in Section 17.18, “Intel® Resource Director Technology (Intel® RDT) Monitoring Features”.
- Features which enable control over shared platform resources are described in Section 17.19, “Intel® Resource Director Technology (Intel® RDT) Allocation Features”.

### 17.1 OVERVIEW OF DEBUG SUPPORT FACILITIES

The following processor facilities support debugging and performance monitoring:

- **Debug exception (#DB)** — Transfers program control to a debug procedure or task when a debug event occurs.
- **Breakpoint exception (#BP)** — See breakpoint instruction (INT3) below.
- **Breakpoint-address registers (DR0 through DR3)** — Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6)** — Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7)** — Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS** — Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register** — Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register** — Generates a debug exception (#DB) after every execution of an instruction.
- **Breakpoint instruction (INT3)** — Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set instruction breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.



- Execution of the breakpoint instruction.
- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

## 17.2 DEBUG REGISTERS

Eight debug registers (see Figure 17-1 for 32-bit operation and Figure 17-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

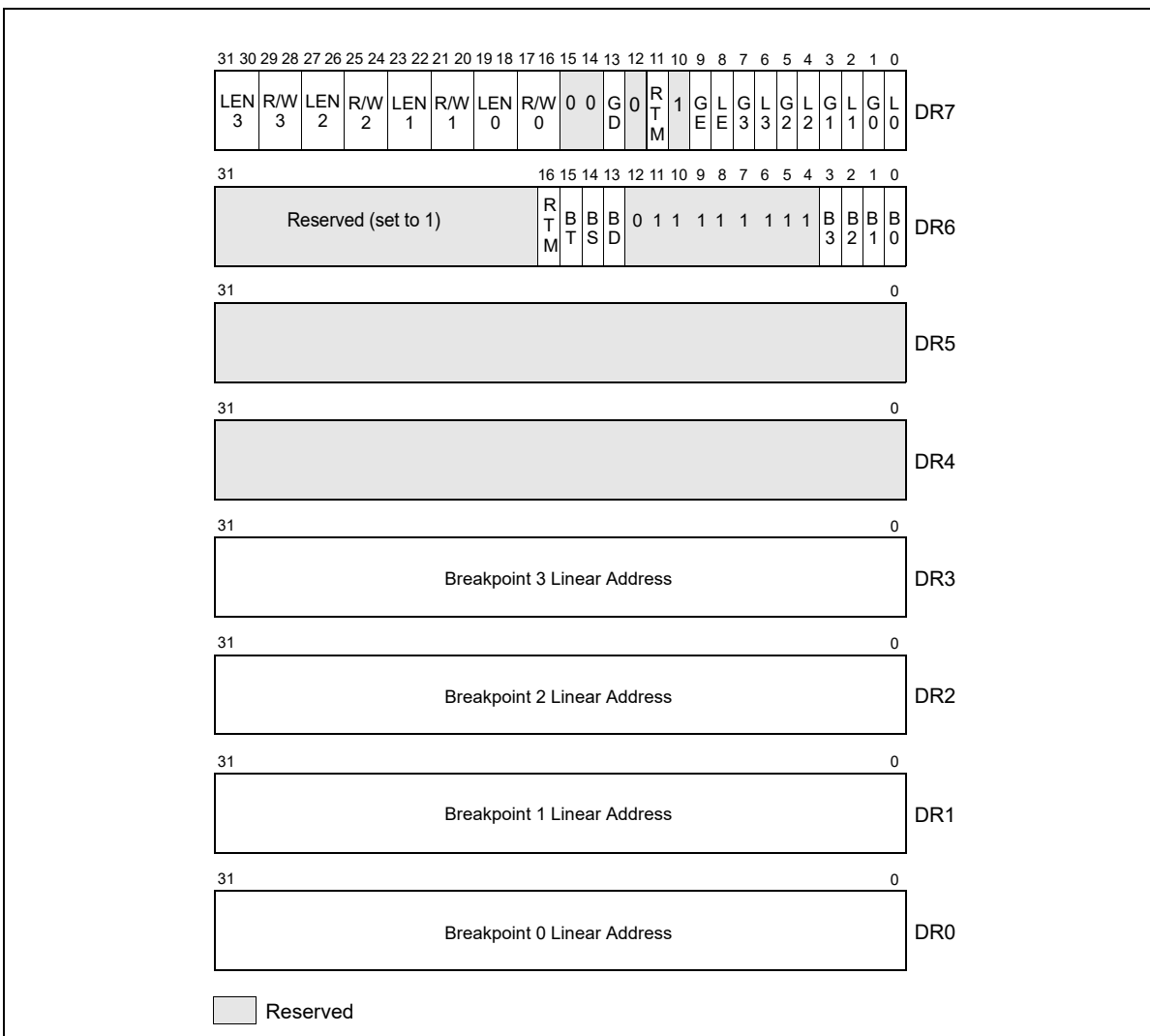


Figure 17-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 through 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 17.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

### 17.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 17-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

### 17.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

### 17.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 17-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3)** — Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the  $LEN_n$  and  $R/W_n$  flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the  $Ln$  or the  $Gn$  flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BD (debug register access detected) flag (bit 13)** — Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 17.2.4, “Debug Control Register (DR7),” for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14)** — Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15)** — Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, “Task-State Segment (TSS),” for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when clear) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

### 17.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 17-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)** — Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated  $L_n$  flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7)** — Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated  $G_n$  flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9)** — This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 17.3.3). This advanced debugging is enabled only if IA32\_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13)** — Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.

- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)** — Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the  $R/W_n$  fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Break on I/O reads or writes.
- 11 — Break on data reads or writes but not instruction fetches.

When the DE flag is clear, the processor interprets the  $R/W_n$  bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

- 00 — Break on instruction execution only.
- 01 — Break on data writes only.
- 10 — Undefined.
- 11 — Break on data reads or writes but not instruction fetches.

- **LENO 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), breakpoint 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). Instruction breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

**Table 17-1. Breakpoint Examples**

Debug Register Setup			
Debug Register	R/Wn	Breakpoint Address	LENn
DR0	R/W0 = 11 (Read/Write)	A0001H	LEN0 = 00 (1 byte)
DR1	R/W1 = 01 (Write)	A0002H	LEN1 = 00 (1 byte)
DR2	R/W2 = 11 (Read/Write)	B0002H	LEN2 = 01) (2 bytes)
DR3	R/W3 = 01 (Write)	C0000H	LEN3 = 11 (4 bytes)
Data Accesses			
Operation		Address	Access Length (In Bytes)
Data operations that trap			
- Read or write		A0001H	1
- Read or write		A0001H	2
- Write		A0002H	1
- Write		A0002H	2
- Read or write		B0001H	4
- Read or write		B0002H	1
- Read or write		B0002H	2
- Write		C0000H	4
- Write		C0001H	2
- Write		C0003H	1
Data operations that do not trap			
- Read or write		A0000H	1
- Read		A0002H	1
- Read or write		A0003H	4
- Read or write		B0000H	2
- Read		C0000H	2
- Read or write		C0004H	4

### 17.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 17-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Break point conditions for 8-byte memory read/writes are supported in all modes.

## 17.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

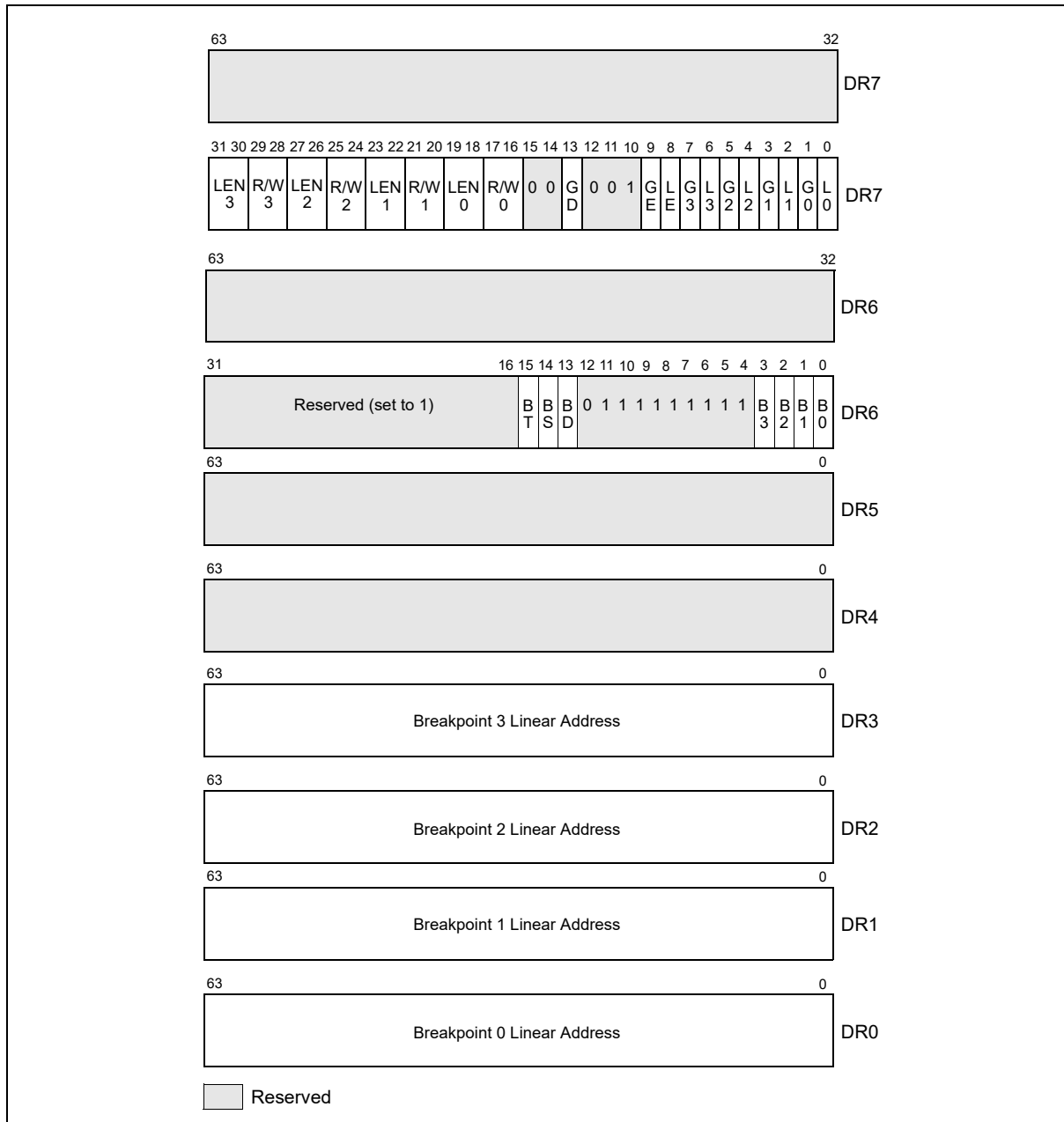


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

### 17.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 17-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 17.3.1.3, “General-Detect Exception Condition”) result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

The INT1 instruction generates a debug exception as a trap. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

See also: Chapter 6, “Interrupt 1—Debug Exception (#DB),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Table 17-2. Debug Exception Conditions**

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 0	Fault
Data write breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 1	Trap
I/O read or write breakpoint, at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 2	Trap
Data read or write (but not instruction fetches), at addresses defined by DR <sub>n</sub> and LEN <sub>n</sub>	B <sub>n</sub> = 1 and (G <sub>n</sub> or L <sub>n</sub> = 1)	R/W <sub>n</sub> = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1	None	Fault
Task switch	BT = 1	None	Trap
INT1 instruction	None	None	Trap

### 17.3.1.1 Instruction-Breakpoint Exception Condition

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

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if an instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint will be suppressed as if EFLAGS.RF were 1 (see the next paragraph and Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, “System Flags and Fields in the EFLAGS Register,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for instruction breakpoints, CS limit violations, and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.

- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - Debug exceptions generated in response to instruction breakpoints
  - Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)
  - Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)
  - Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

### 17.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

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

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

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

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

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

If an execution of the MOV or POP instruction loads the SS register and encounters a data breakpoint, the resulting debug exception is delivered after completion of the next instruction (the one after the MOV or POP).

Any pending data or I/O breakpoints are lost upon delivery of an exception. For example, if a machine-check exception (#MC) occurs following an instruction that encounters a data breakpoint (but before the resulting debug exception is delivered), the data breakpoint is lost. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, the data breakpoint is lost if the next instruction causes a fault.

Delivery of events due to INT *n*, INT3, or INTO does not cause a loss of data breakpoints. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, and the next instruction is software interrupt (INT *n*, INT3, or INTO), a debug exception (#DB) resulting from a data breakpoint will be delivered after the transition to the software-interrupt handler. The #DB handler should account for the fact that the #DB may have been delivered



after a invocation of a software-interrupt handler, and in particular that the CPL may have changed between recognition of the data breakpoint and delivery of the #DB.

### 17.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

### 17.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n*, INT3, and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

If an occurrence of the MOV or POP instruction loads the SS register executes with EFLAGS.TF = 1, no single-step debug exception occurs following the MOV or POP instruction.

### 17.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

## 17.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP)." Debuggers use breakpoint exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

### 17.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify transactional regions (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32\_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does not resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered. (A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

## 17.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt and exception recording:

- Section 17.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)”
- Section 17.6, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture”
- Section 17.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem”
- Section 17.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”
- Section 17.11, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture”
- Section 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture”
- Section 17.14, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
- Section 17.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)”
- Section 17.16, “Last Branch, Interrupt, and Exception Recording (P6 Family Processors)”

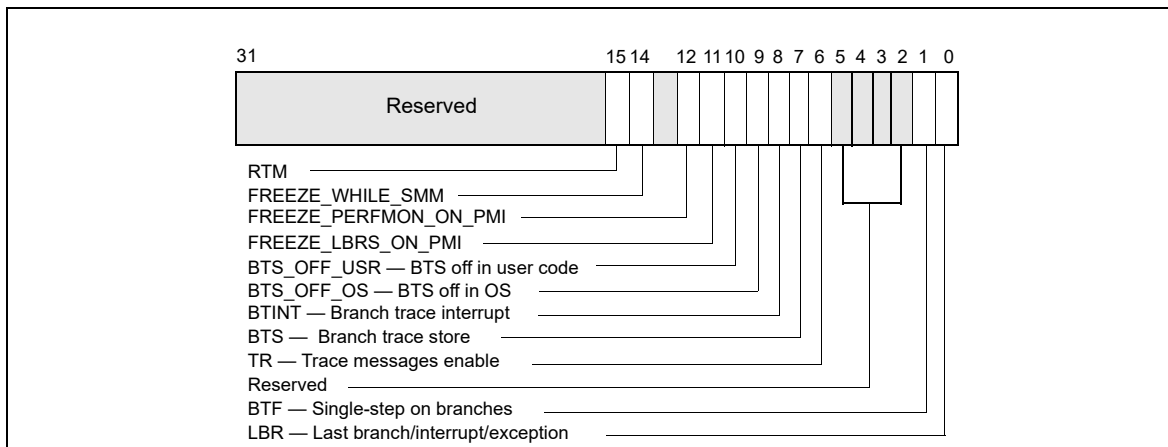
The following subsections of Section 17.4 describe common features of profiling branches. These features are generally enabled using the IA32\_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR\_DEBUGCTLA, MSR\_DEBUGCTLB, MSR\_DEBUGCTL).

### 17.4.1 IA32\_DEBUGCTL MSR

The IA32\_DEBUGCTL MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32\_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 17.5.1, “LBR Stack” (Intel® Core™2 Duo and Intel® Atom™ Processor Family) and Section 17.9.1, “LBR Stack” (processors based on Intel® Microarchitecture code name Nehalem).
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.
- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bit 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.



**Figure 17-3. IA32\_DEBUGCTL MSR for Processors based on Intel Core microarchitecture**

- **BTS\_OFF\_OS (branch trace off in privileged code) flag (bit 9)** — When set, BTS or BTM is skipped if CPL is 0. See Section 17.13.2.
- **BTS\_OFF\_USR (branch trace off in user code) flag (bit 10)** — When set, BTS or BTM is skipped if CPL is greater than 0. See Section 17.13.2.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — When set, the LBR stack is frozen on a hardware PMI request (e.g. when a counter overflows and is configured to trigger PMI). See Section 17.4.7 for details.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — When set, the performance counters (IA32\_PMCx and IA32\_FIXED\_CTRx) are frozen on a PMI request. See Section 17.4.7 for details.
- **FREEZE\_WHILE\_SMM (bit 14)** — If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32\_PERF\_GLOBAL\_CTRL, save a copy of the content of IA32\_DEBUGCTL and disable LBR, BTF,

TR, and BTS fields of IA32\_DEBUGCTL before transferring control to the SMI handler. Subsequently, the enable bits of IA32\_PERF\_GLOBAL\_CTRL will be set to 1, the saved copy of IA32\_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. Note that system software must check if the processor supports the IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM control bit.

IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 18.8 for details of detecting the presence of IA32\_PERF\_CAPABILITIES MSR.

- **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 17.3.3.

## 17.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32\_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32\_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

## 17.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32\_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.<sup>1</sup> This mechanism allows a debugger to single-step on control transfers caused by branches. This “branch single stepping” helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

## 17.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32\_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 17.4.2, “Monitoring Branches, Exceptions, and Interrupts.”

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

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32\_DEBUGCTL MSR (or the equivalent MSR\_DEBUGCTLA, MSR\_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, “Task-State Segment (TSS).”

### 17.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

### 17.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32\_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 17.4.9.2, "Setting Up the DS Save Area." for additional details.

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

### 17.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if `CPUID.01H:ECX[bit 4] = 1`.

The CPL-qualified branch trace mechanism is described in Section 17.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, `BTS_OFF_USR` (bit 10) and `BTS_OFF_OS` (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

### 17.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e. `CPUID.0AH:EAX[7:0] > 1`). These capabilities provides the following interface in IA32\_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by `CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0]`, two flavors are supported:
  - Legacy `Freeze_LBR_on_PMI` is supported for `ArchPerfMonVerID <= 3` and `ArchPerfMonVerID > 1`. If `IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1`, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32\_DEBUGCTL. Software must then re-enable `IA32_DEBUGCTL.LBR` to resume recording branches. When using this feature, software should be careful about writes to IA32\_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - Streamlined `Freeze_LBR_on_PMI` is supported for `ArchPerfMonVerID >= 4`. If `IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1`, the processor behaves as follows:
    - sets `IA32_PERF_GLOBAL_STATUS.LBR_Frz = 1` to disable recording, but does not change the LBR bit (bit 0) in IA32\_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI (bit 12)** — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by `CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0]`, two flavors are supported:

- Legacy Freeze\_Perfmon\_on\_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32\_DEBUGCTL.Freeze\_Perfmon\_On\_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32\_PERF\_GLOBAL\_CTRL MSR (see Figure 18-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 18.6.2.1, “Fixed-function Performance Counters”). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32\_PERF\_GLOBAL\_CTRL before leaving a PMI service routine to continue counter operation.
- Streamlined Freeze\_Perfmon\_on\_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
  - sets IA32\_PERF\_GLOBAL\_STATUS.CTR\_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32\_PERF\_GLOBAL\_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

- A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  - For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32\_PERFEVTSELx register.
  - For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR\_PERF\_FIXED\_CTR\_CTRL register (see Figure 18-1) or IA32\_FIXED\_CTR\_CTRL MSR (see Figure 18-2).
- The PEBS buffer is almost full and reaches the interrupt threshold.
- The BTS buffer is almost full and reaches the interrupt threshold.

Table 17-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeza\_Perfmon\_On\_PMI interface.

**Table 17-3. Legacy and Streamlined Operation with Freeze\_Perfmon\_On\_PMI = 1, Counter Overflowed**

Legacy Freeze_Perfmon_On_PMI	Streamlined Freeze_Perfmon_On_PMI	Comment
Processor freezes the counters on overflow	Processor freezes the counters on overflow	Unchanged
Processor clears IA32_PERF_GLOBAL_CTRL	Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ	
Handler reads IA32_PERF_GLOBAL_STATUS (0x38E) to examine which counter(s) overflowed	<b>mask</b> = RDMSR(0x38E)	Similar
Handler services the PMI	Handler services the PMI	Unchanged
Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL (0x390)	Handler writes <b>mask</b> 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

**Table 17-4. LBR Stack Size and TOS Pointer Range (Contd.)**

DisplayFamily_DisplayModel	Size of LBR Stack	Component of an LBR Entry	Range of TOS Pointer
06_4EH, 06_5EH, 06_8EH, 06_9EH, 06_55H, 06_66H, 06_7AH	32	FROM_IP, TO_IP, LBR_INFO <sup>1</sup>	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 employ 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.

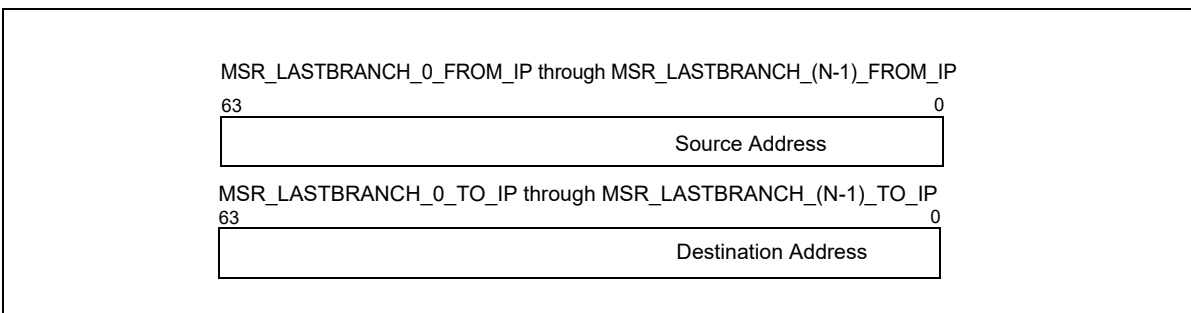


Figure 17-4. 64-bit Address Layout of LBR MSR

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.

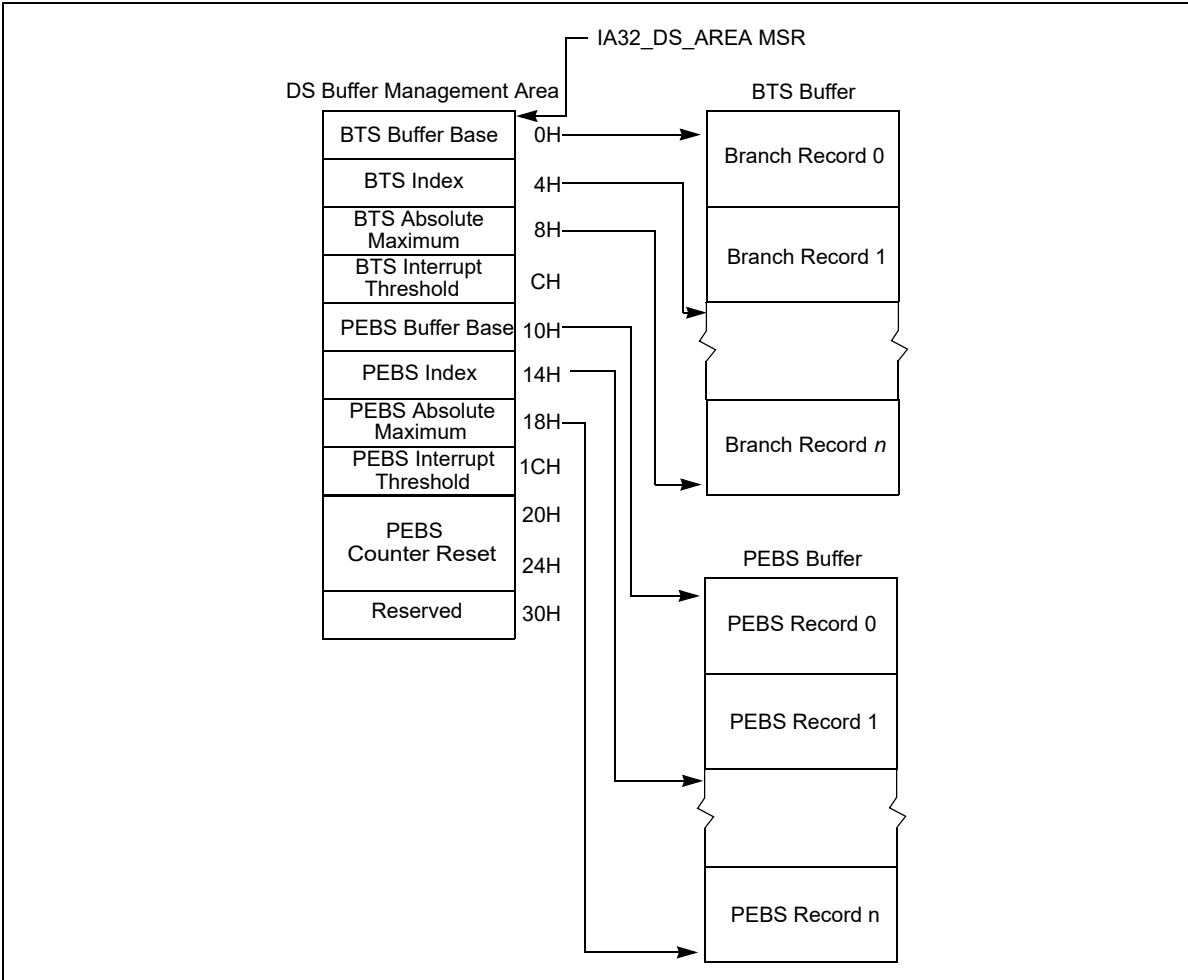


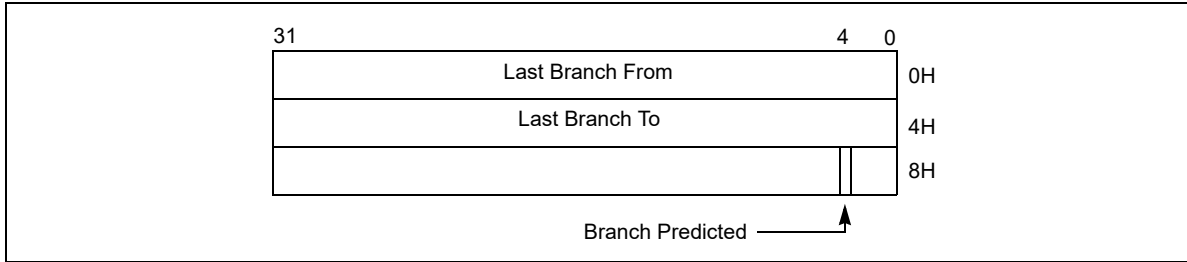
Figure 17-5. DS Save Area Example<sup>1</sup>

**NOTES:**

1. This example represents the format for a system that supports PEBS on only one counter.

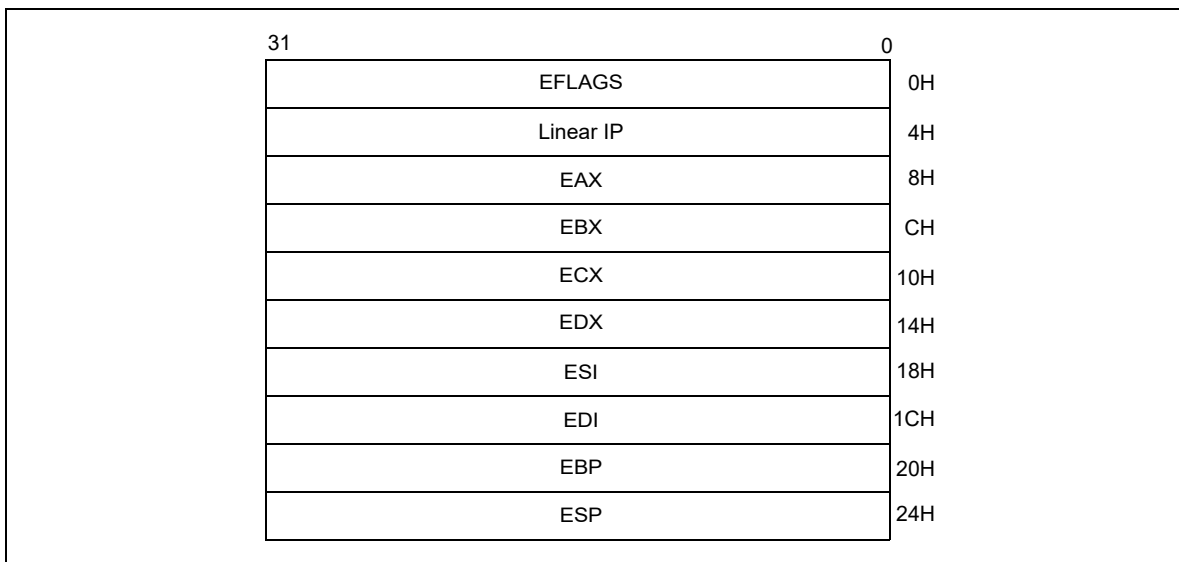
Figure 17-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **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.

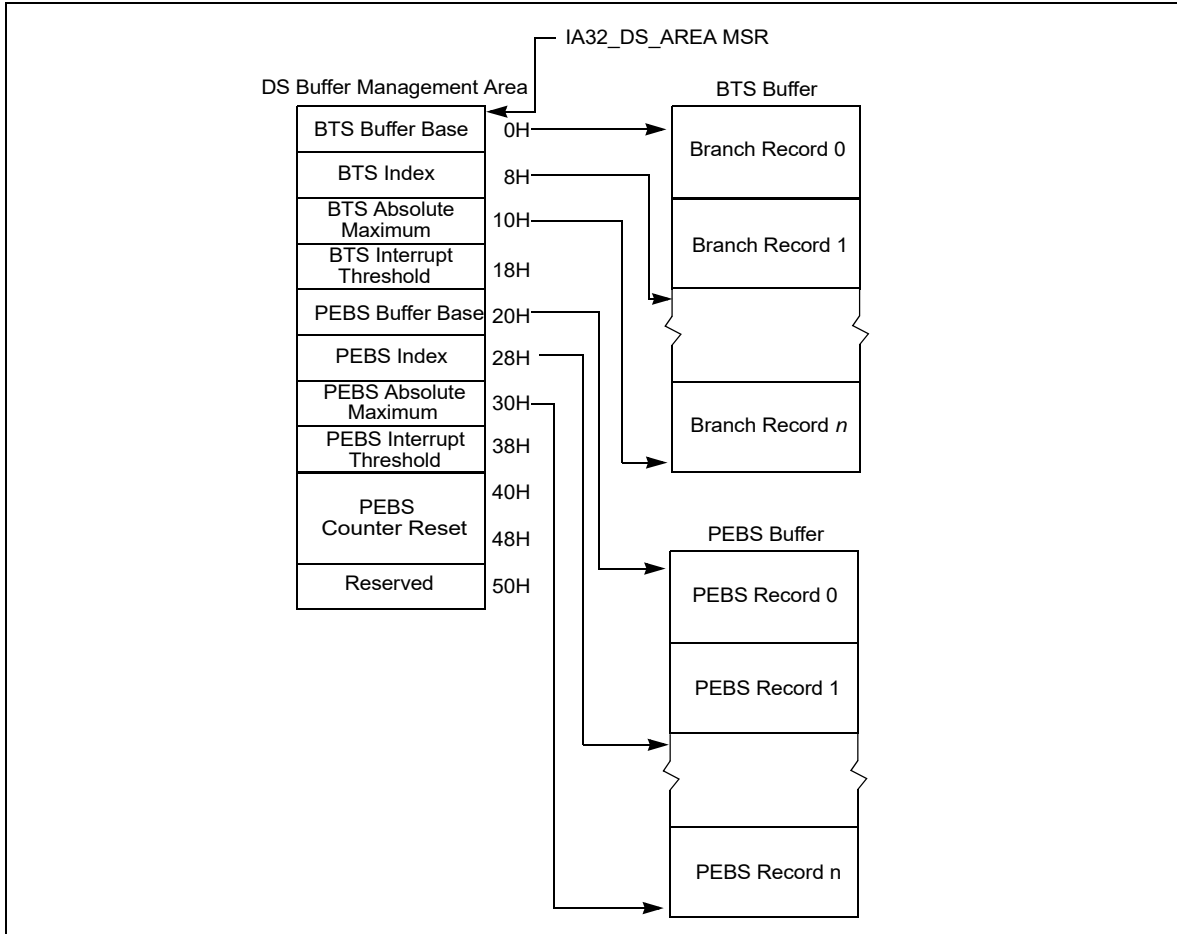


Figure 17-8. IA-32e Mode DS Save Area Example<sup>1</sup>

**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).

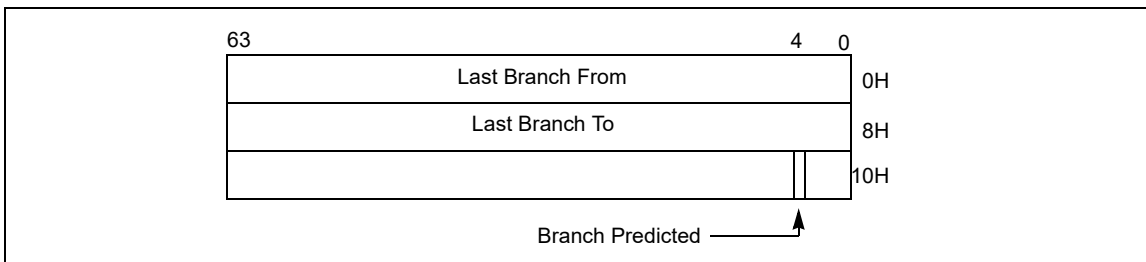
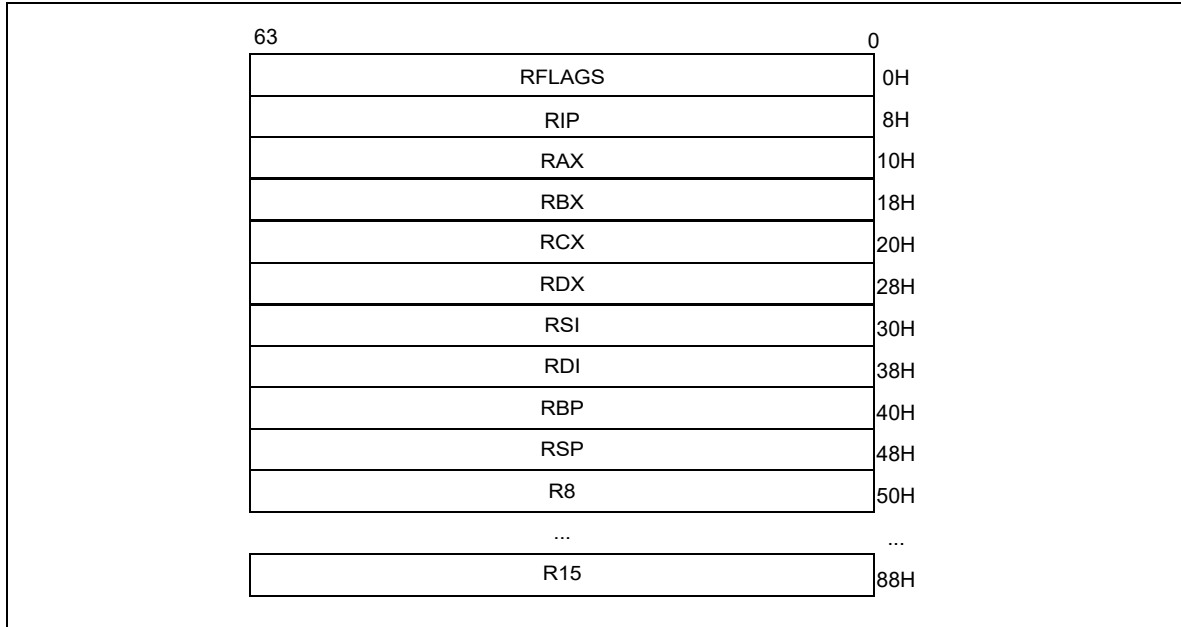


Figure 17-9. 64-bit Branch Trace Record Format



**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.

- Write an interrupt service routine to handle the interrupt. See Section 17.4.9.5, “Writing the DS Interrupt Service Routine.”

The following restrictions should be applied to the DS save area.

- The three DS save area sections should be allocated from a non-paged pool, and marked accessed and dirty. It is the responsibility of the operating system to keep the pages that contain the buffer present and to mark them accessed and dirty. The implication is that the operating system cannot do “lazy” page-table entry propagation for these pages.
- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.
- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.
- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.
- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.
- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.
- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.
- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.
- The DS save area is expected to be used only on systems with an enabled APIC. The LVT Performance Counter entry in the APIC 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:

- 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.
- 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).
- Clear the BTINT flag in the corresponding IA32\_DEBUGCTL (or MSR\_DEBUGCTLA MSR; or MSR\_DEBUGCTLB) if a circular BTS buffer is desired.

## NOTES

If the buffer size is set to less than the minimum allowable value (i.e. BTS absolute maximum < 1 + size of BTS record), the results of BTS is undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

### 17.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS\_OFF\_OS, BTS\_OFF\_USR, and BTINT. The encoding of these five bits are shown in Table 17-6.

**Table 17-6. CPL-Qualified Branch Trace Store Encodings**

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

### 17.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.

BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.

- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32\_DEBUGCTL (or MSR\_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR\_PEBS\_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.
- The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.



- After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.
- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the counters to count via IA32\_PERF\_GLOBAL\_CTRL/IA32\_PERF\_GLOBAL\_OVF\_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR's ENABLE bit on processor based on Intel NetBurst microarchitecture).
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

## 17.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL® ATOM™ PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

- **Debug Trace and Branch Recording Control** — The IA32\_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-3 for the MSR layout.
- **Last branch record (LBR) stack** — There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.5.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts**
  - See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  - 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE\_LBRS\_ON\_PMI flag is set.
- **Branch trace messages** — See Section 17.4.4.
- **Last exception records** — See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.5.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze\_LBRS\_On\_PMI operation.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze\_Perfmon\_On\_PMI operation.
- **FREEZE\_WHILE\_SMM (bit 14)** — FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 17.4.1.

### 17.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 40H) through MSR\_LASTBRANCH\_3\_FROM\_IP (address 43H) store source addresses
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 60H) through MSR\_LASTBRANCH\_3\_TO\_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 40H) through MSR\_LASTBRANCH\_7\_FROM\_IP (address 47H) store source addresses
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 60H) through MSR\_LASTBRANCH\_7\_TO\_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM\_IP/TO\_IP MSRS may differ between processors. Software should query IA32\_PERF\_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

### 17.5.2 LBR Stack in Intel Atom Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT. The layout of MSR\_LBR\_SELECT is described in Table 17-11.

## 17.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Processors based on the Goldmont microarchitecture extend the capabilities described in Section 17.5.2 with the following enhancements:

- Supports new LBR format encoding 00110b in IA32\_PERF\_CAPABILITIES[5:0].
- Size of LBR stack increased to 32. Each entry includes MSR\_LASTBRANCH\_x\_FROM\_IP (address 0x680..0x69f) and MSR\_LASTBRANCH\_x\_TO\_IP (address 0x6c0..0x6df).
- LBR call stack filtering supported. The layout of MSR\_LBR\_SELECT is described in Table 17-13.
- Elapsed cycle information is added to MSR\_LASTBRANCH\_x\_TO\_IP. Format is shown in Table 17-7.
- Misprediction info is reported in the upper bits of MSR\_LASTBRANCH\_x\_FROM\_IP. MISRPRED bit format is shown in Table 17-8.
- Streamlined Freeze\_LBRs\_On\_PMI operation; see Section 17.12.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 17.12.3.

**Table 17-7. MSR\_LASTBRANCH\_x\_TO\_IP for the Goldmont Microarchitecture**

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to” address. See Section 17.4.8.1 for address format.
Cycle Count (Saturating)	63:48	R/W	Elapsed core clocks since last update to the LBR stack.

## 17.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 17.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32\_PERF\_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of three MSRs:
  - MSR\_LASTBRANCH\_x\_FROM\_IP, the layout is simplified, see Table 17-9.
  - MSR\_LASTBRANCH\_x\_TO\_IP, the layout is the same as Table 17-9.
  - MSR\_LBR\_INFO\_x, stores branch prediction flag, TSX info, and elapsed cycle data. Layout is the same as Table 17-16.

## 17.8 LAST BRANCH, INTERRUPT AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR\_LASTBRANCH\_0\_FROM\_IP (address 680H) through MSR\_LASTBRANCH\_7\_FROM\_IP (address 687H) store source addresses.
  - MSR\_LASTBRANCH\_0\_TO\_IP (address 6C0H) through MSR\_LASTBRANCH\_7\_TO\_IP (address 6C7H) store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT. The layout of MSR\_LBR\_SELECT is described in Table 17-11.

The address format written in the FROM\_IP/TO\_IP MSRS may differ between processors. Software should query IA32\_PERF\_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

## 17.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

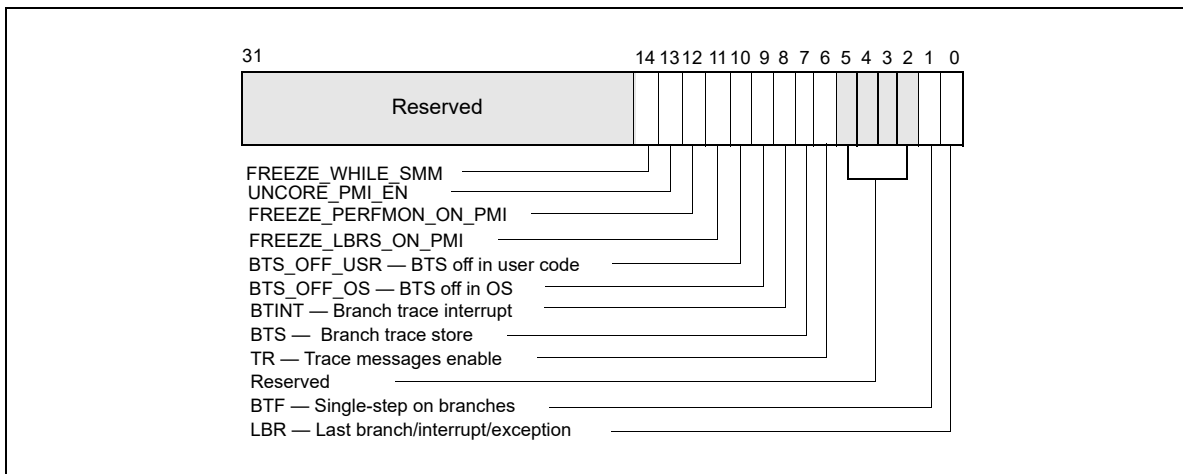
The processors based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and adds additional capabilities:

- **Debug Trace and Branch Recording Control** — The IA32\_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-11 for the MSR layout.
- **Last branch record (LBR) stack** — There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.9.1.

- **Monitoring and single-stepping of branches, exceptions, and interrupts** — See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
- **Branch trace messages** — The IA32\_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 17.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records** — See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.6 and Section 17.4.5.
- **FREEZE\_LBRS\_ON\_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze\_LBRs\_On\_PMI operation.
- **FREEZE\_PERFMON\_ON\_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze\_Perfmon\_On\_PMI operation.
- **UNCORE\_PMI\_EN (bit 13)** — When set, this logical processor is enabled to receive an counter overflow interrupt form the uncore.
- **FREEZE\_WHILE\_SMM (bit 14)** — FREEZE\_WHILE\_SMM is supported if IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is reporting 1. See Section 17.4.1.

Processors based on Intel microarchitecture code name Nehalem provide additional capabilities:

- **Independent control of uncore PMI** — The IA32\_DEBUGCTL MSR provides a bit field (see Figure 17-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.
- **LBR filtering** — Processors based on Intel microarchitecture code name Nehalem support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR\_LBR\_SELECT.



**Figure 17-11. IA32\_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem**

## 17.9.1 LBR Stack

Processors based on Intel microarchitecture code name Nehalem provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 17-8 and Table 17-9.

**Table 17-8. MSR\_LASTBRANCH\_x\_FROM\_IP**

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch from” address. See Section 17.4.8.1 for address format.
SIGN_EXt	62:48	R/W	Signed extension of bit 47 of this register.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

**Table 17-9. MSR\_LASTBRANCH\_x\_TO\_IP**

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to” address. See Section 17.4.8.1 for address format
SIGN_EXt	63:48	R/W	Signed extension of bit 47 of this register.

Processors based on Intel microarchitecture code name Nehalem have an LBR MSR Stack as shown in Table 17-10.

**Table 17-10. LBR Stack Size and TOS Pointer Range**

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_1AH	16	0 to 15

## 17.9.2 Filtering of Last Branch Records

MSR\_LBR\_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e. all branches will be captured. MSR\_LBR\_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR\_LBR\_SELECT is shown in Table 17-11.

**Table 17-11. MSR\_LBR\_SELECT for Intel microarchitecture code name Nehalem**

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

## 17.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem”, apply to processors based on Intel microarchitecture code name Sandy Bridge. For processors based on Intel microarchitecture code name Ivy Bridge, the same holds true.

One difference of note is that MSR\_LBR\_SELECT is shared between two logical processors in the same core. In Intel microarchitecture code name Sandy Bridge, each logical processor has its own MSR\_LBR\_SELECT. The filtering semantics for “Near\_ind\_jmp” and “Near\_rel\_jmp” has been enhanced, see Table 17-12.

**Table 17-12. MSR\_LBR\_SELECT for Intel® microarchitecture code name Sandy Bridge**

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

## 17.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”, apply to next generation processors based on Intel microarchitecture code name Haswell.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR\_LBR\_SELECT.EN\_CALLSTACK[bit 9]; see Table 17-13. If MSR\_LBR\_SELECT.EN\_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 17.10.

**Table 17-13. MSR\_LBR\_SELECT for Intel® microarchitecture code name Haswell**

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.

**Table 17-13. MSR\_LBR\_SELECT for Intel® microarchitecture code name Haswell**

Bit Field	Bit Offset	Access	Description
FAR_BRANCH	8	R/W	When set, do not capture far branches
EN_CALLSTACK <sup>1</sup>	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 [CPLUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPLUID family 0FH, model 03H]. See also: Table 17-17, Figure 17-12, and Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- Last exception record — See Section 17.13.3, “Last Exception Records.”

### 17.13.1 MSR\_DEBUGCTLA MSR

The MSR\_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-12 shows the flags in the MSR\_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example, when an instruction or data breakpoint or a single-step trap occurs). See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches.”
- **TR (trace message enable) flag (bit 2)** — When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages.”

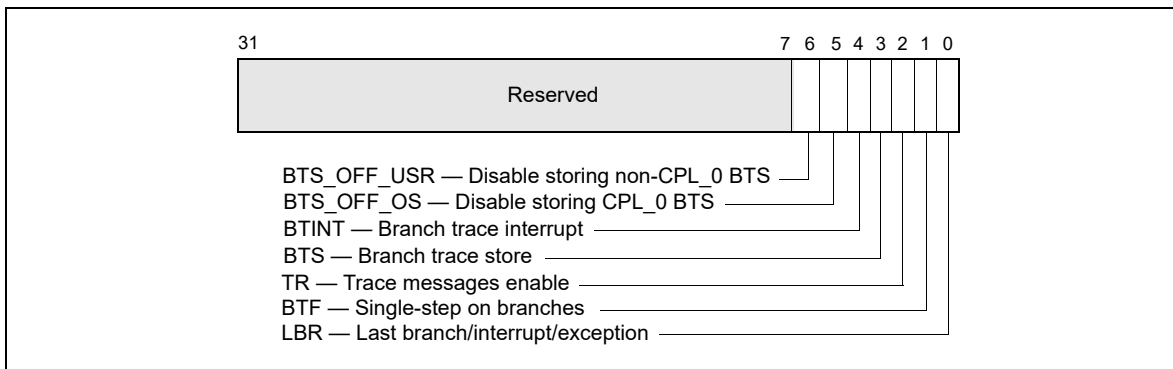


Figure 17-12. MSR\_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **BTS (branch trace store) flag (bit 3)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 4)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS).”
- **BTS\_OFF\_OS (disable ring 0 branch trace store) flag (bit 5)** — When set, enables the BTS facilities to skip sending/logging CPL\_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTS\_OFF\_USR (disable ring 0 branch trace store) flag (bit 6)** — When set, enables the BTS facilities to skip sending/logging non-CPL\_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

**NOTE**

The initial implementation of BTS\_OFF\_USR and BTS\_OFF\_OS in MSR\_DEBUGCTLA is shown in Figure 17-12. The BTS\_OFF\_USR and BTS\_OFF\_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

**17.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture**

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR\_LASTBRANCH\_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches its maximum value, it wraps around to 0. See Table 17-17 and Figure 17-12.

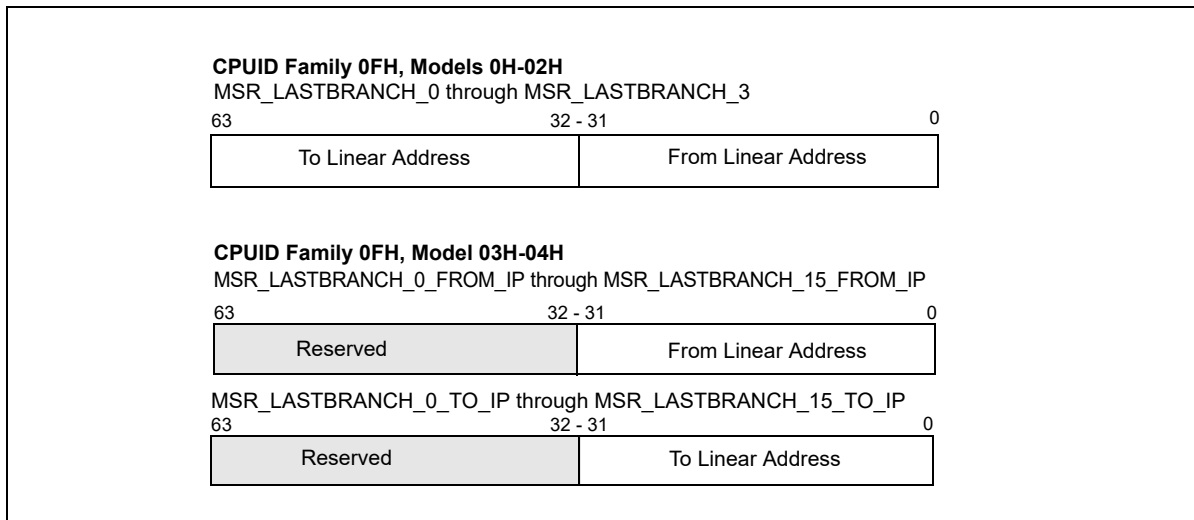
**Table 17-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family**

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR\_LASTBRANCH\_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 17-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the “from” and “to” instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the “from” address is the address of the branch instruction and the “to” address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the “from” address is the return instruction pointer (RIP) saved for the interrupt and the “to” address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the “from” address is the linear address of the instruction that caused the exception to be generated and the “to” address is the address of the first instruction in the exception handler routine.



**Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family**

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

### 17.13.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors provide two MSRs (the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors. The MSR\_LER\_TO\_LIP and MSR\_LER\_FROM\_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

## 17.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

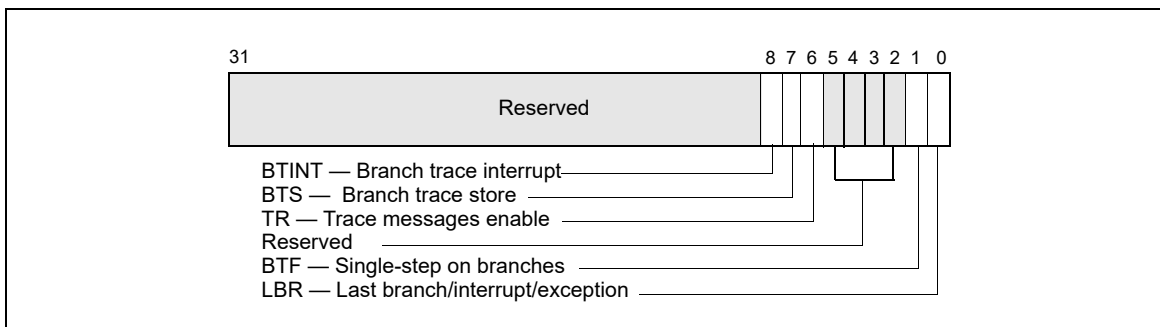
- **IA32\_DEBUGCTL MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32\_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-14 for the layout and the entries below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the “Last Branch Record (LBR) Stack” below.
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism

allows single-stepping the processor on taken branches. See Section 17.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.

- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.

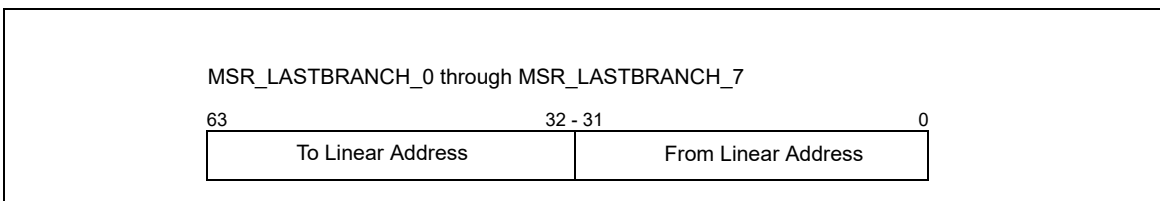


**Figure 17-14. IA32\_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors**

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, “Branch Trace Store (BTS).”
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR\_LASTBRANCH\_0 through MSR\_LASTBRANCH\_7); bits 31-0 hold the ‘from’ address, bits 63-32 hold the ‘to’ address (MSR addresses start at 40H). See Figure 17-15.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR\_LER\_TO\_LIP and the MSR\_LER\_FROM\_LIP MSRs) that duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

For details, see Section 17.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture,” and Section 2.19, “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.
  - **PBi (performance monitoring/breakpoint pins) flags (bits 5-2)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BPi# pin when a breakpoint match occurs. When a PBi flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages,” for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
  - **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.

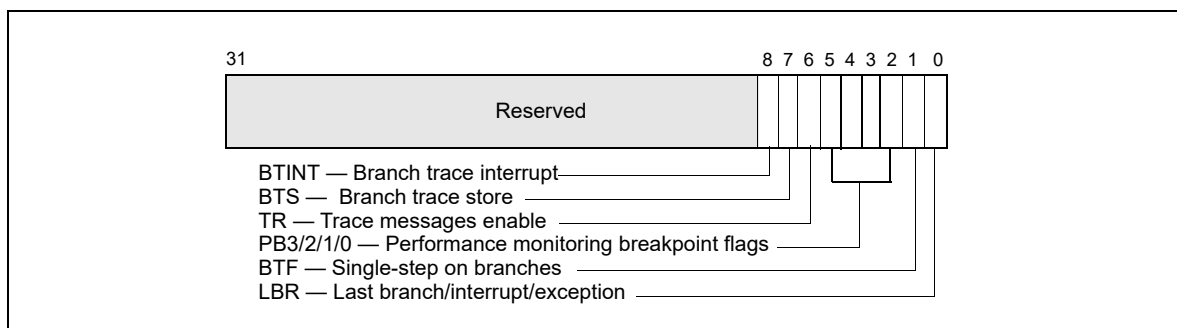
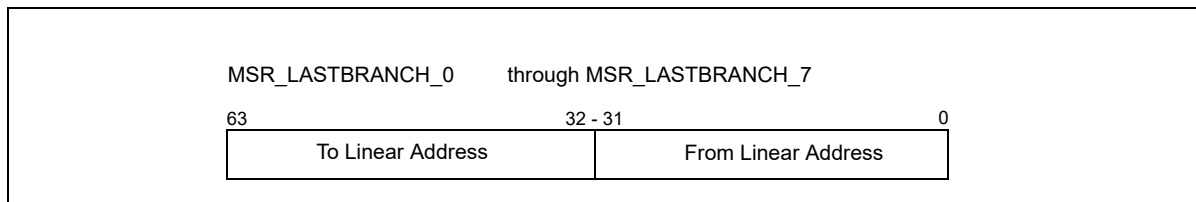


Figure 17-16. MSR\_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, “Branch Trace Store (BTS).”

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 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: DEBUGCTLMR, 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 DEBUGCTLMR Register

The version of the DEBUGCTLMR 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 DEBUGCTLMR 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.

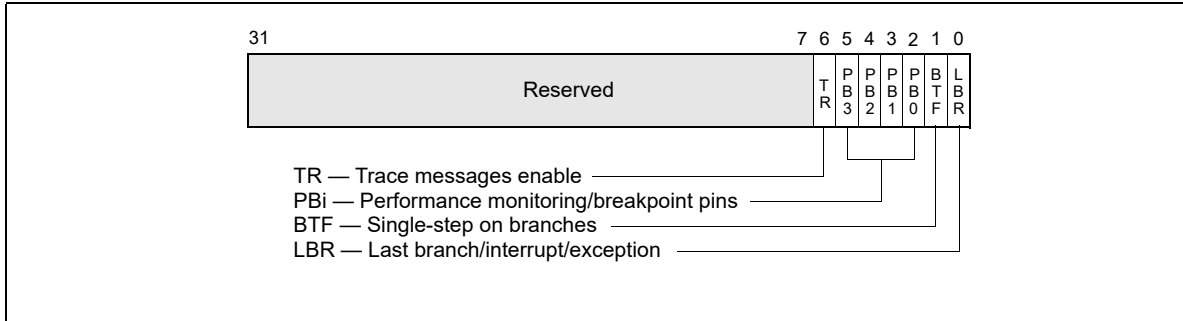


Figure 17-18. DEBUGCTMSR Register (P6 Family Processors)

- **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.”
- **P<sub>B</sub>*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<sub>*i*</sub># pin when a breakpoint match occurs. When a P<sub>B</sub>*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 DEBUGCTMSR 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.80000011H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).

User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.

### 17.17.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32\_TIME\_STAMP\_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way that ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32\_TSC\_ADJUST MSR (address 3BH). Like the IA32\_TIME\_STAMP\_COUNTER MSR, the IA32\_TSC\_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32\_TSC\_ADJUST MSR as follows:

- On RESET, the value of the IA32\_TSC\_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32\_TIME\_STAMP\_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32\_TSC\_ADJUST MSR.
- If an execution of WRMSR to the IA32\_TSC\_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32\_TSC\_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32\_TIME\_STAMP\_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32\_TSC\_ADJUST MSR on each logical processor.

Processor support for the IA32\_TSC\_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC\_ADJUST (bit 1).

### 17.17.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

$$\text{TSC\_Value} = (\text{ART\_Value} * \text{CPUID.15H:EBX[31:0]}) / \text{CPUID.15H:EAX[31:0]} + K$$

Where 'K' is an offset that can be adjusted by a privileged agent<sup>2</sup>.

When ART hardware is reset, both invariant TSC and K are also reset.

## 17.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel® Xeon® processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case

2. IA32\_TSC\_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time

focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

### 17.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of **Resource Monitoring IDs (RMIDs)**. Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32\_PQR\_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32\_QM\_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. Before reading back monitored data software must configure an event selection MSR (IA32\_QM\_EVTSEL) to specify which metric is to be reported, and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

### 17.18.2 Enabling Monitoring: Usage Flow

Figure 17-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.

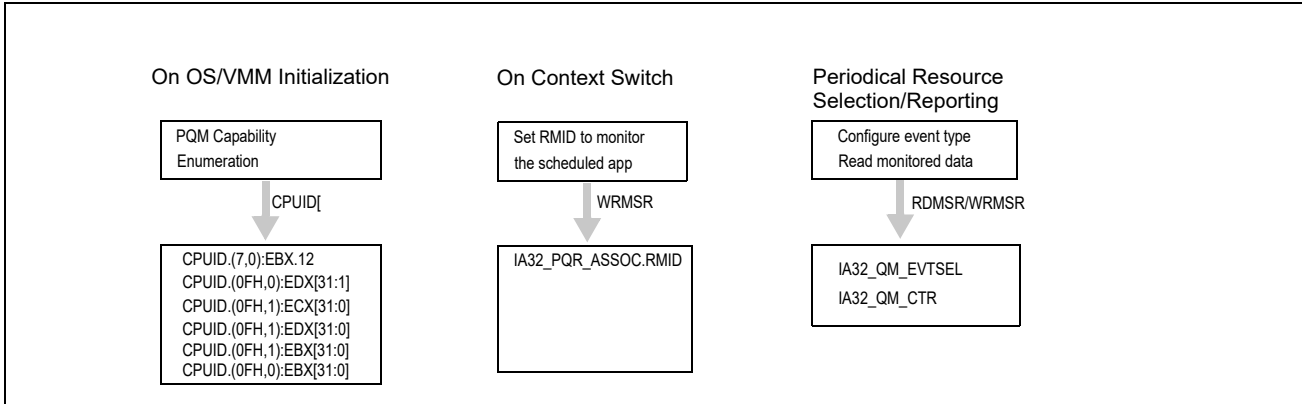


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

### 17.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CUID instruction with EAX = 07H, ECX = 0H as input. If CUID.(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:

- CUID 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 CUID with EAX=0 to discover the "cpuid\_maxLeaf" supported in the processor;
2. If cpuid\_maxLeaf >= 7, then execute CUID with EAX=7, ECX= 0 to verify CUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.
5. If CUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CUID.(EAX=0FH, ECX=0):EDX.

### 17.18.4 Monitoring Resource Type and Capability Enumeration

CUID 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 CUID 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.

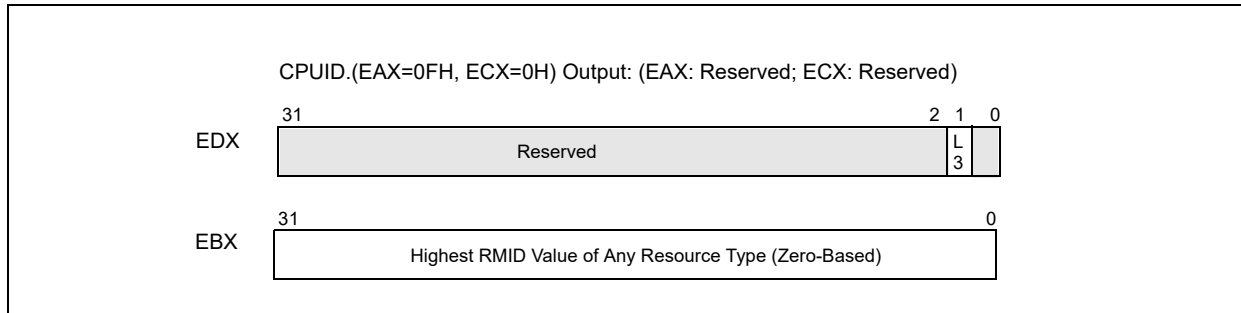


Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

### 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 MSR using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.

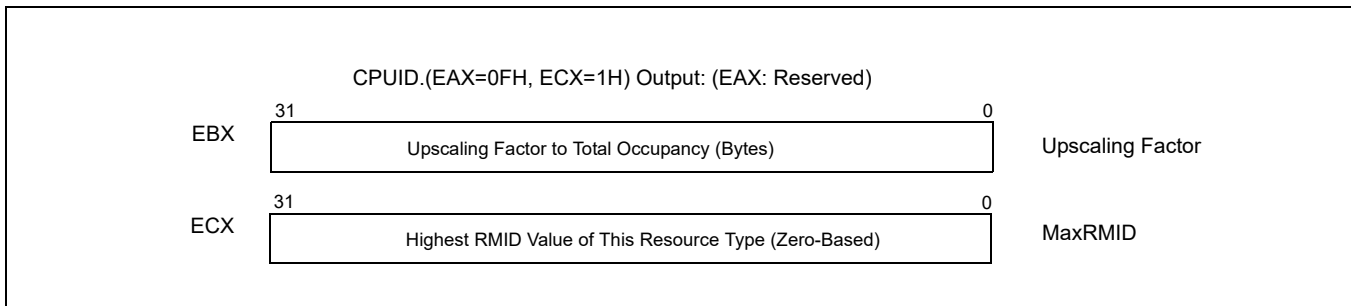


Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H) )

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 17-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 17-22 and Table 17-18) that software must program with IA32\_QM\_EVTSEL in order to retrieve event data. After software configures IA32\_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32\_QM\_CTR. The raw numerical value reported from IA32\_QM\_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 17-21.

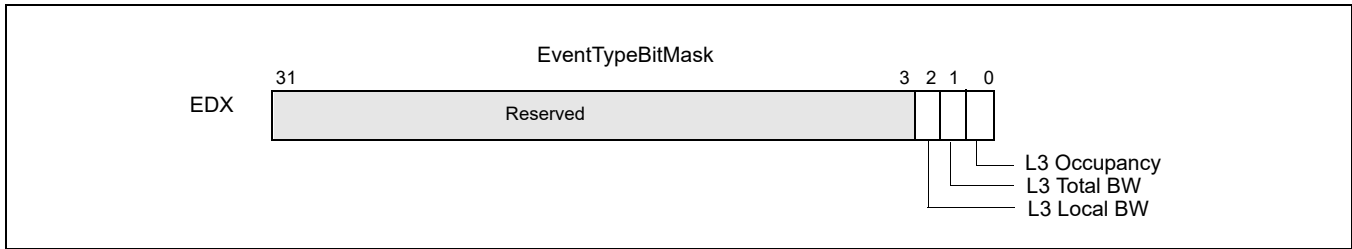


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H) )

### 17.18.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX would return with only bit 0 set. The corresponding event ID can be looked up from Table 17-18. The L3 occupancy data accumulated in IA32\_QM\_CTR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

### 17.18.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits will be set in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.
- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth (for instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction).

The corresponding Event ID can be looked up from Table 17-18. The L3 bandwidth data accumulated in IA32\_QM\_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.

Table 17-18. Monitoring Supported Event IDs

Event Type	Event ID	Context
L3 Cache Occupancy	01H	Cache Monitoring Technology
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM
Reserved	All other event codes	N/A

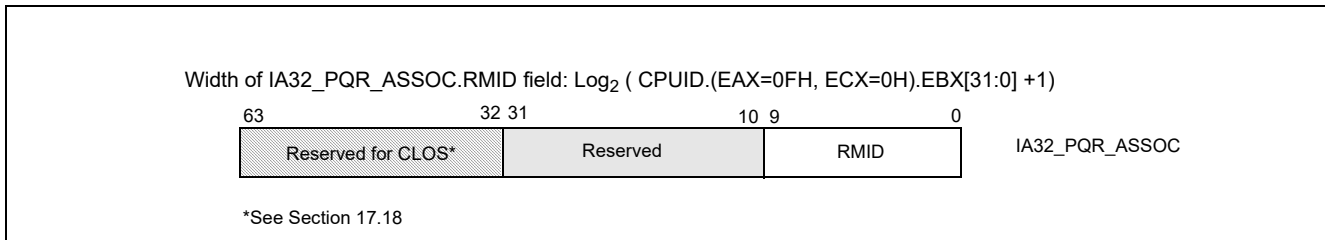
### 17.18.6 Monitoring Resource RMID Association

After Monitoring and sub-features has been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that

RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32\_PQR\_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32\_PQR\_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 17-23. Software specifies the active RMID to monitor in the IA32\_PQR\_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from Ceil ( $\log_2 (1 + \text{CPUID}.\text{EAX}=0\text{FH}, \text{ECX}=0):\text{EBX}[31:0])$ ). The value of IA32\_PQR\_ASSOC after power-on is 0.



**Figure 17-23. IA32\_PQR\_ASSOC MSR**

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32\_PQR\_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g. RMID 31) may only be supported by one resource type but not by another resource type.

### 17.18.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g. no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32\_QM\_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 17-24. Bits IA32\_QM\_EVTSEL.EvtID (bits 7:0) specify an event code of a supported resource type for hardware to report monitored data associated with IA32\_QM\_EVTSEL.RMID (bits 41:32). Software can configure IA32\_QM\_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32\_QM\_EVTSEL.RMID matches that of IA32\_PQR\_ASSOC.RMID. Supported event codes for the IA32\_QM\_EVTSEL register are shown in Table 17-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

Software can program an RMID / Event ID pair into the IA32\_QM\_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 17.18.5, which covers feature-specific details.



Thread access to the IA32\_QM\_EVTSEL and IA32\_QM\_CTR MSR pair should be serialized to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32\_QM\_CTR.

- IA32\_QM\_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32\_QM\_EVTSEL, then IA32\_QM\_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32\_QM\_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32\_QM\_CTR.data (bits 61:0) should be ignored. Therefore, IA32\_QM\_CTR.data (bits 61:0) is valid only if bit 63 and 62 are both clear. For Cache Monitoring Technology, software can convert IA32\_QM\_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.

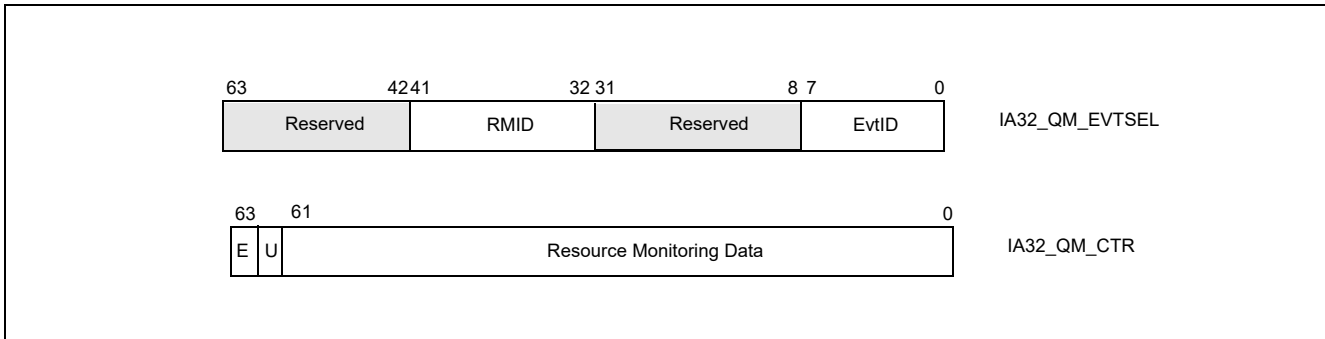


Figure 17-24. IA32\_QM\_EVTSEL and IA32\_QM\_CTR MSRs

### 17.18.8 Monitoring Programming Considerations

Figure 17-23 illustrates how system software can program IA32\_QOSEVTSEL and IA32\_QM\_CTR to perform resource monitoring.

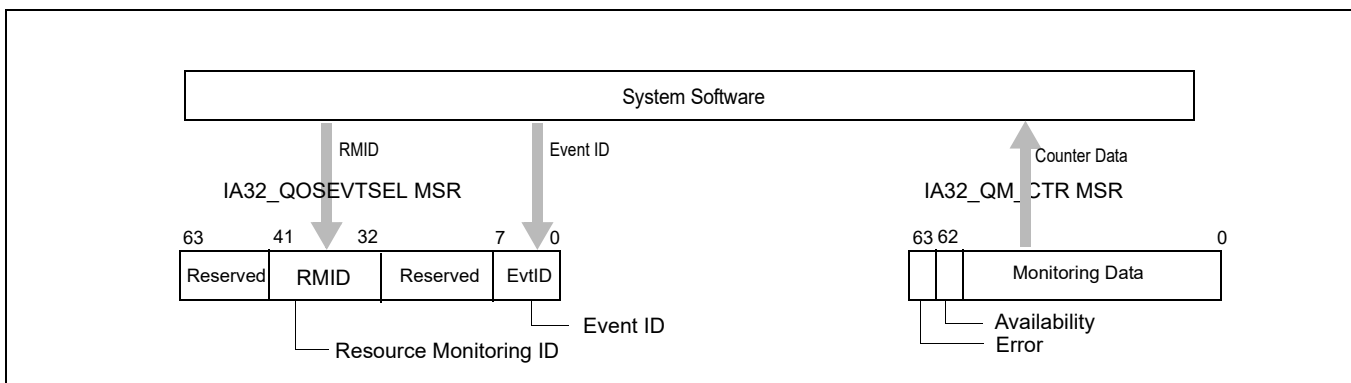


Figure 17-25. Software Usage of Cache Monitoring Resources

Though the field provided in IA32\_QM\_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used will be proportional to the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any future changes to counter width will be enumerated to software.

### 17.18.8.1 Monitoring Dynamic Configuration

Both the IA32\_QM\_EVTSEL and IA32\_PQR\_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- An RMID exceeding the maxRMID is used.

### 17.18.8.2 Monitoring Operation With Power Saving Features

Note that some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.

### 17.18.8.3 Monitoring Operation with Other Operating Modes

The states in IA32\_PQR\_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler's data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

### 17.18.8.4 Monitoring Operation with RAS Features

In general the Reliability, Availability and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses these may be tracked and may influence the shared resource monitoring counter values.

## 17.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and a subset of communication-focused processors in the Intel Xeon E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Certain Intel Atom processors also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter. The CAT and CDP capabilities, where architecturally supported, may be detected and enumerated in software using the *CPUID* instruction, as described in this chapter.

The Intel Xeon Processor Scalable Family introduces the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

### 17.19.1 Introduction to Cache Allocation Technology (CAT)

Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.

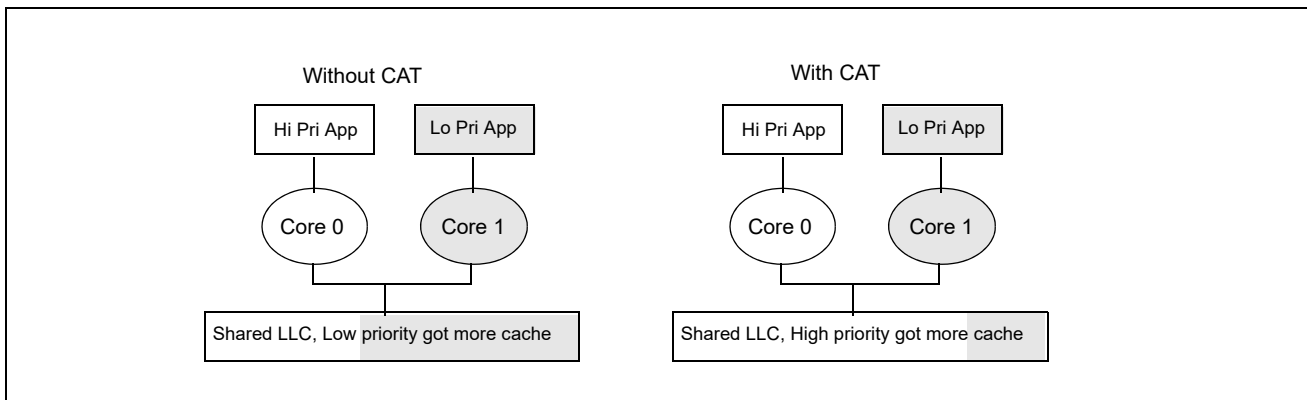
Software can determine which levels are supported in a given platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size,
- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks,
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs, and
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.

The Cache Allocation Technology feature enables more cache resources (i.e. cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 17-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSES, VMMs, containers and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e. OS/VMM) to support such resource control. Note that in Figure 17-26 the L3 Cache is shown as an example resource.



**Figure 17-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications**

### 17.19.2 Cache Allocation Technology Architecture

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS or CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32\_PQR\_ASSOC MSR or PQR) to allow the OS/VMM to specify a COS when an application, thread or VM is scheduled.

The usage of Classes of Service (COS) are consistent across resources and a COS may have multiple resource control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of COS tags per resource for instance, the COS management overhead is constant. Cache allocation for the indicated application/thread/container/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32\_resourceType\_MASK\_n MSRs, where resourceType indicates a resource type (e.g. "L3" for the L3 cache) and "n" indicates a COS number.

The basic ingredients of Cache Allocation Technology are as follows:

- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled,
- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a COS to an executing software thread (i.e. associating the active CR3 of a logical processor with the COS in IA32\_PQR\_ASSOC),
- Implementation-dependent mechanisms to indicate which COS is associated with a memory access and to enforce the cache allocation on a per COS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bit length of the capacity mask available generally depends on the configuration of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported COS may vary for each resource type, and these details can be enumerated via CPUID.

	M7	M6	M5	M4	M3	M2	M1	M0	
<b>COS0</b>	A	A	A	A	A	A	A	A	<b>Default Bitmask</b>
<b>COS1</b>	A	A	A	A	A	A	A	A	
<b>COS2</b>	A	A	A	A	A	A	A	A	
<b>COS3</b>	A	A	A	A	A	A	A	A	
	M7	M6	M5	M4	M3	M2	M1	M0	
<b>COS0</b>	A	A	A	A	A	A	A	A	<b>Overlapped Bitmask</b>
<b>COS1</b>					A	A	A	A	
<b>COS2</b>							A	A	
<b>COS3</b>								A	
	M7	M6	M5	M4	M3	M2	M1	M0	
<b>COS0</b>	A	A	A	A					<b>Isolated Bitmask</b>
<b>COS1</b>					A	A			
<b>COS2</b>							A		
<b>COS3</b>								A	

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.

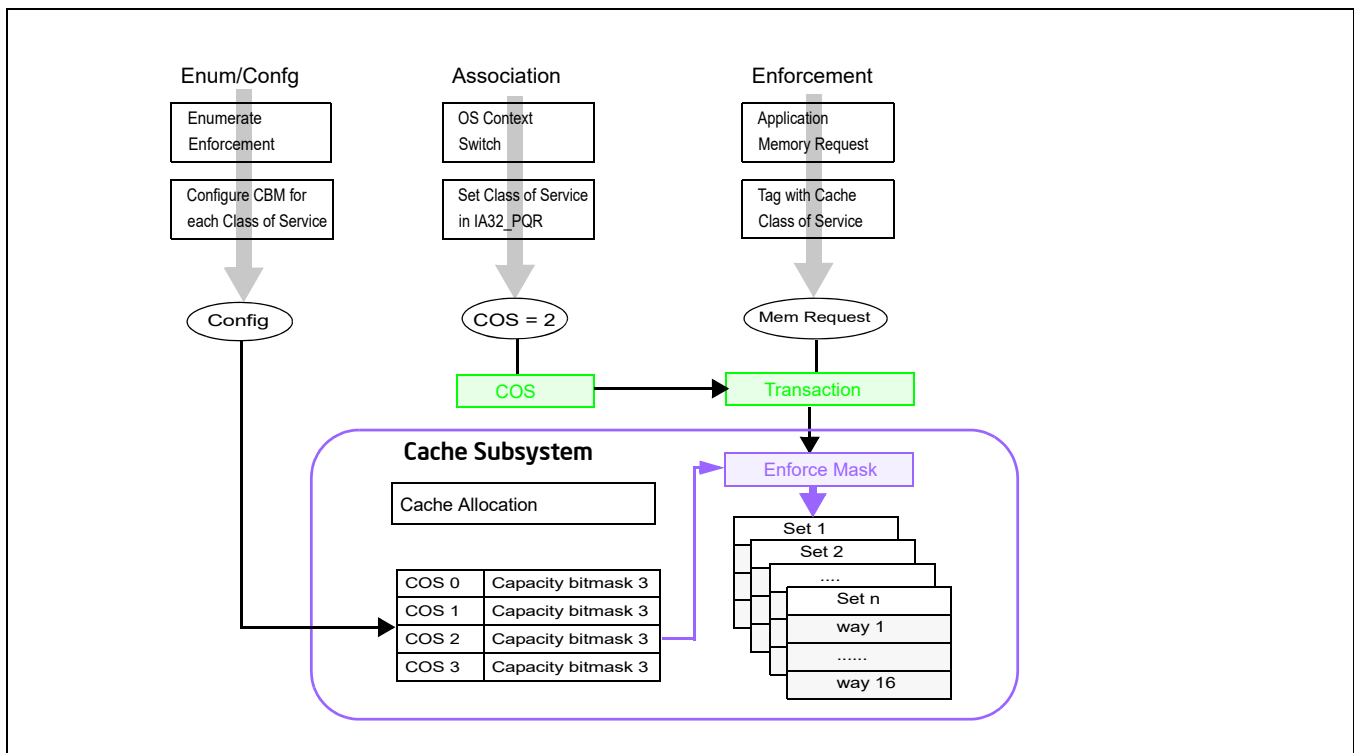


Figure 17-28. Class of Service and Cache Capacity Bitmasks

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.

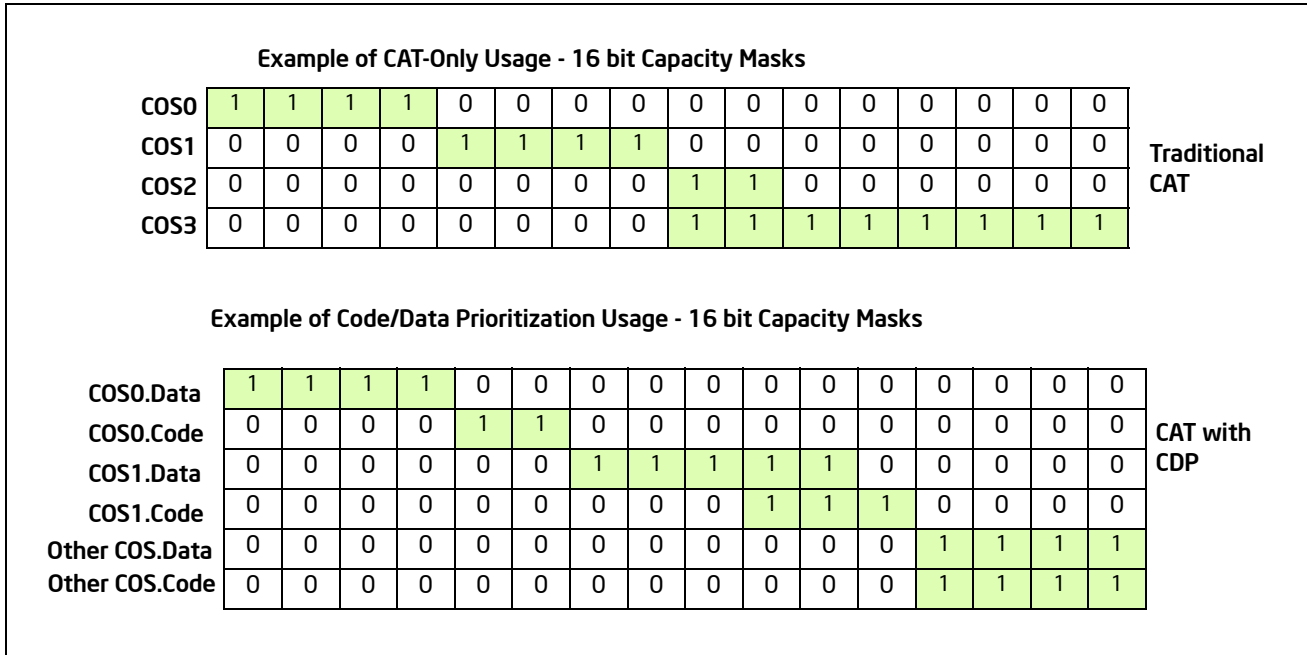
By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled,

- 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.

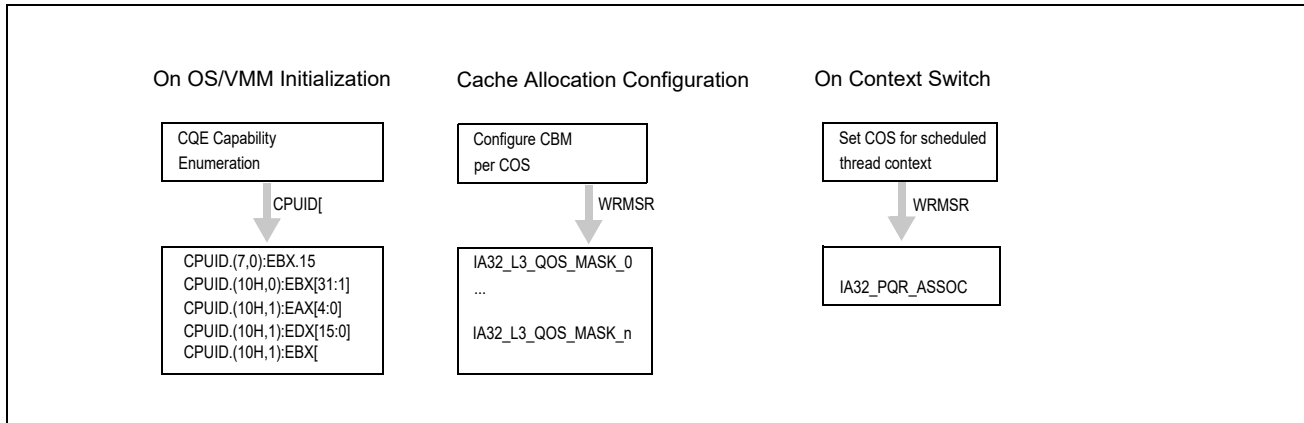


Figure 17-30. Cache Allocation Technology Usage Flow

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.

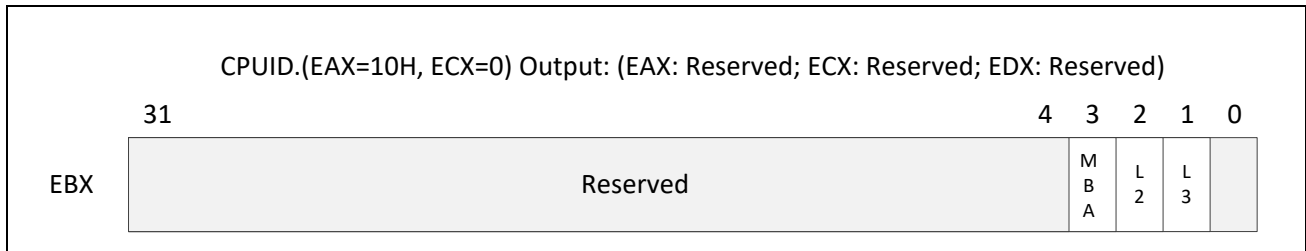
### 17.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

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

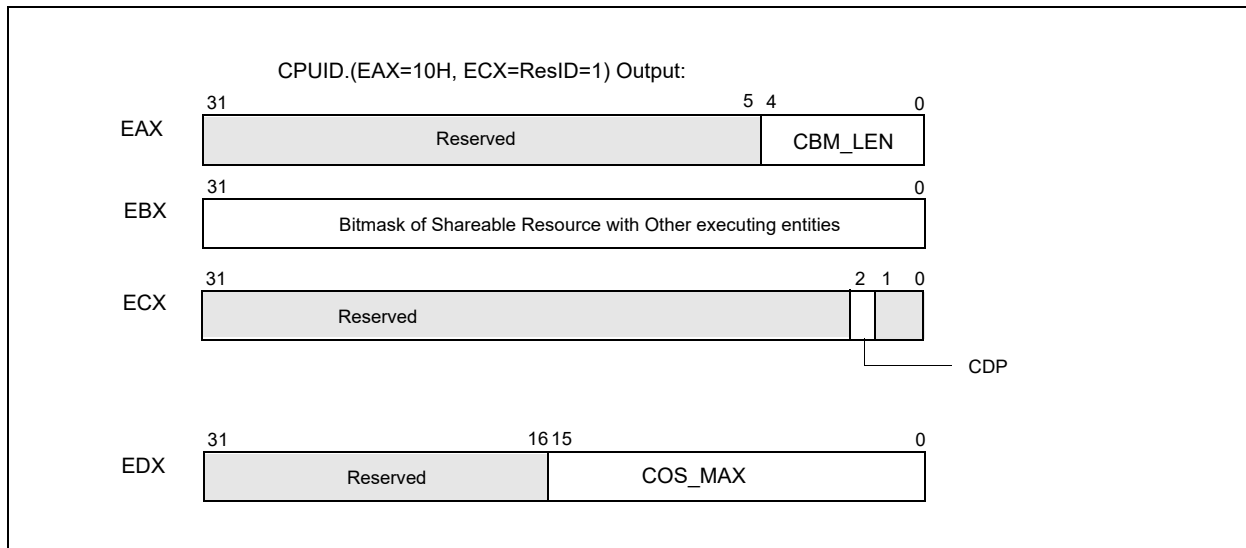


CPUID.(EAX=10H, ECX=0):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of 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:

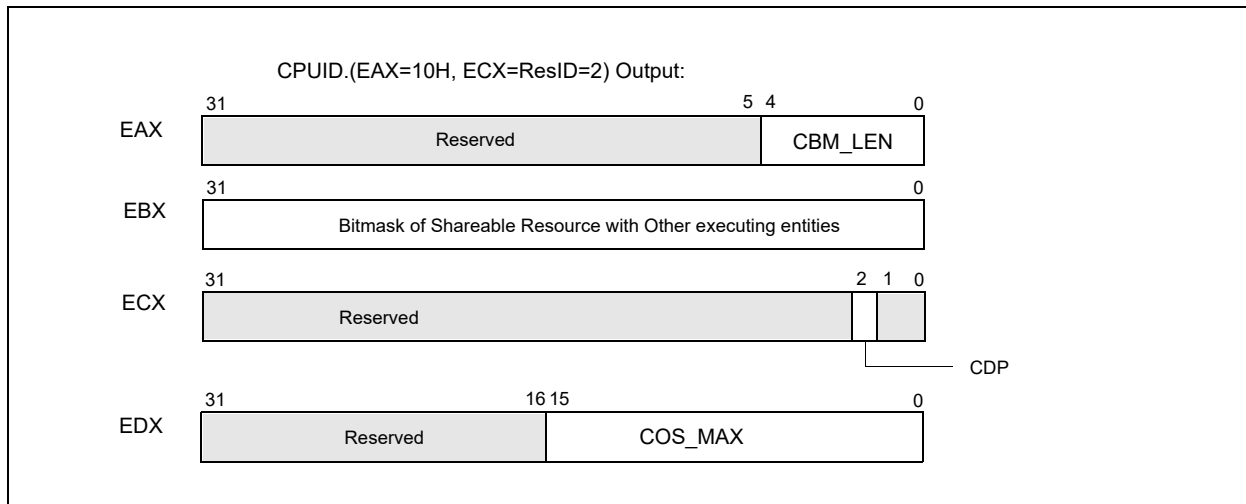


**Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration**

- 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.

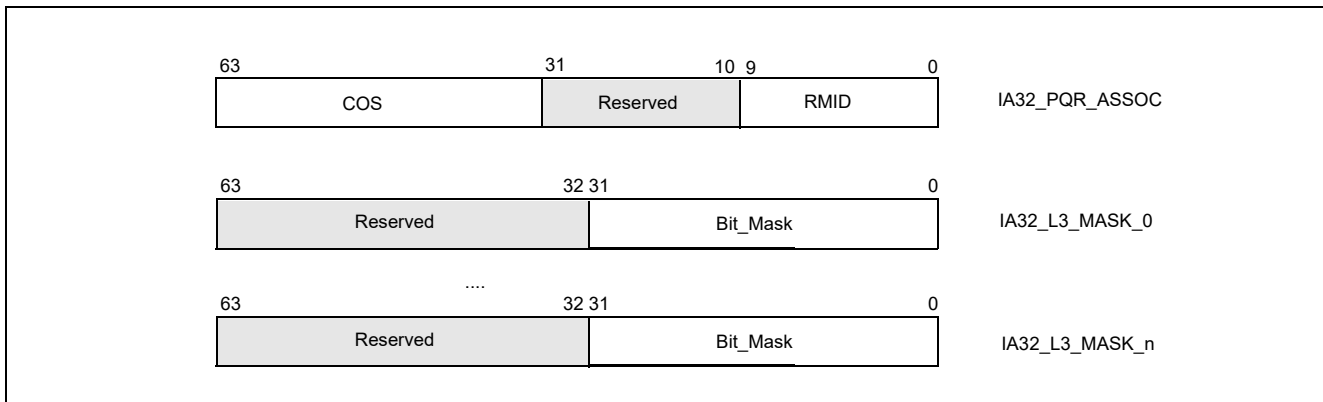


Figure 17-34. IA32\_PQR\_ASSOC, IA32\_L3\_MASK\_n MSRs

- 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.

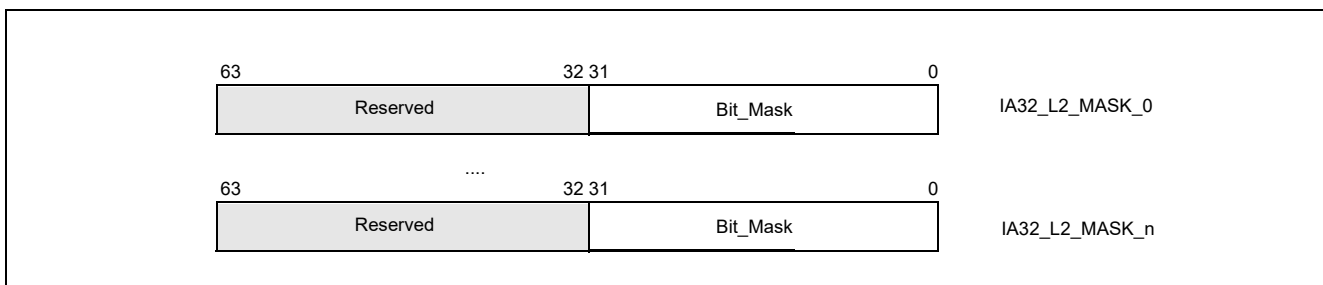


Figure 17-35. IA32\_L2\_MASK\_n MSRs

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).

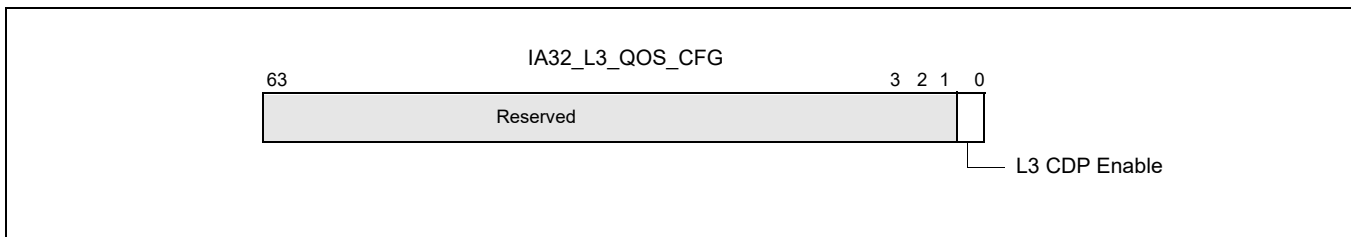


Figure 17-36. Layout of IA32\_L3\_QOS\_CFG

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).

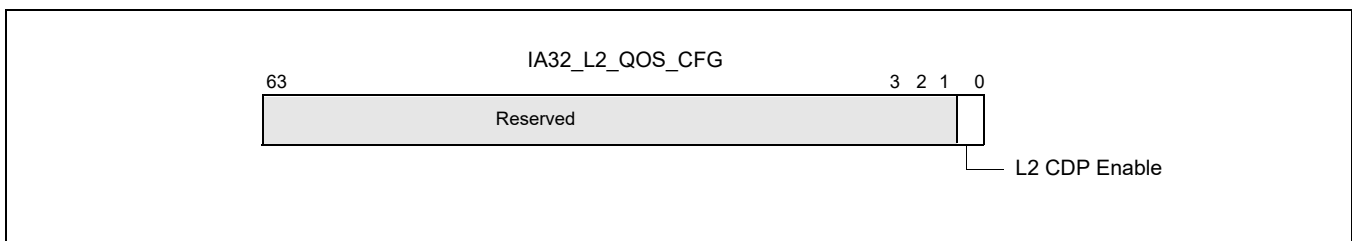
Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 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).

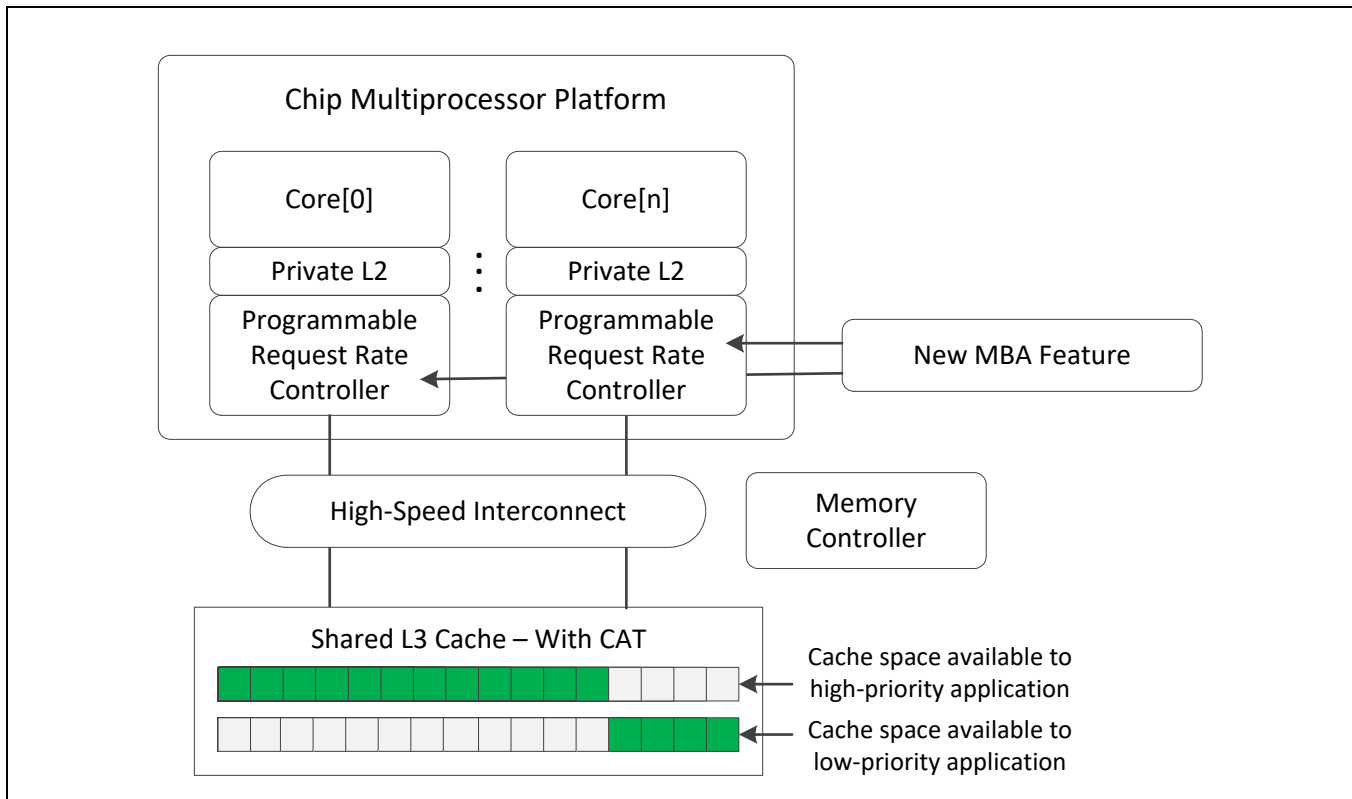
#### 17.19.7 Introduction to Memory Bandwidth Allocation

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.

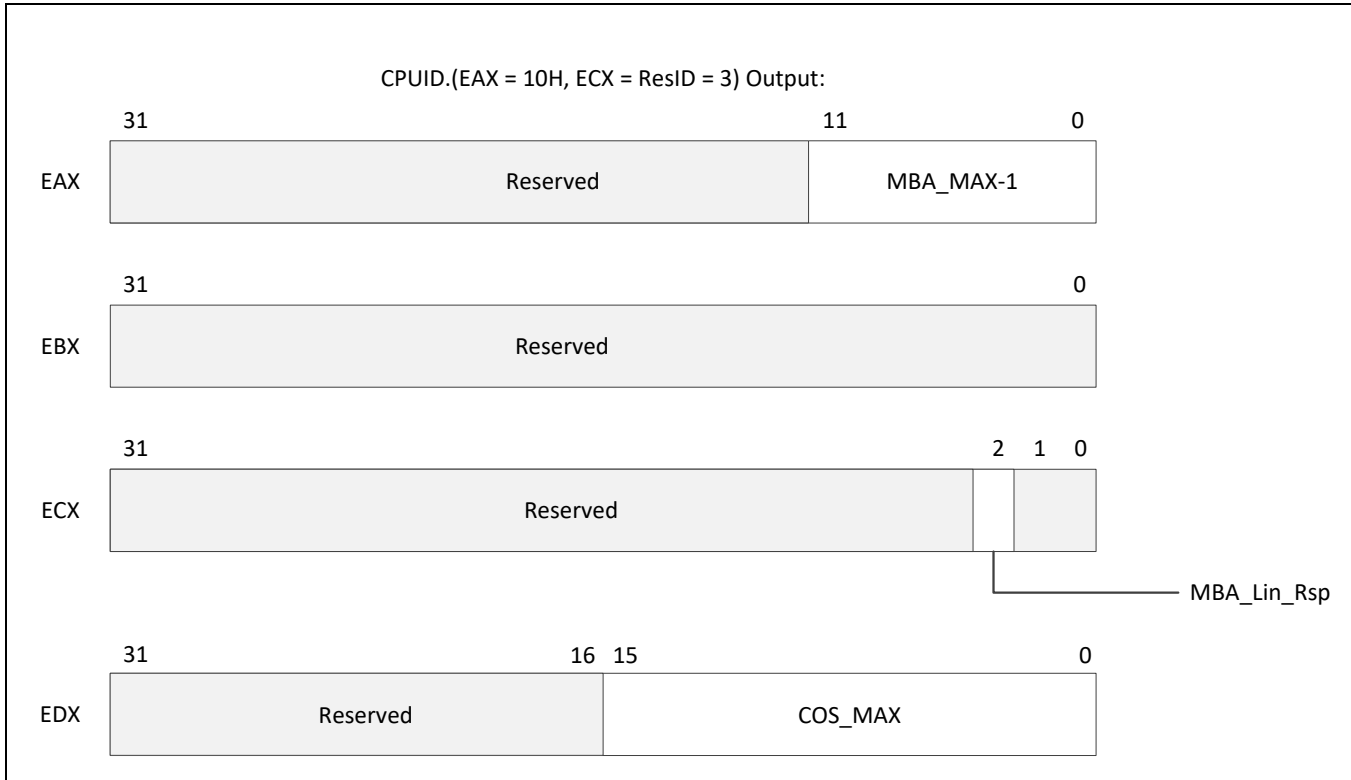
The presence of any of the RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software 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.

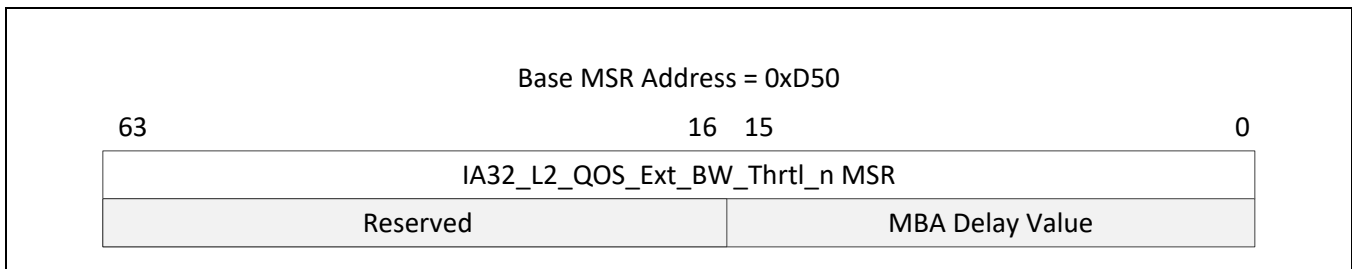
**Table 17-20. MBA Delay Value MSRs**

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 inadvertent throttling of the high-priority thread.

## 12. Updates to Chapter 18, Volume 3B

Change bars show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes to this chapter: Update to Table 18-75 "Nominal Core Crystal Clock Frequency", adding Intel® Xeon® W Processor Family.

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 MSR. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

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

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)." Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they are listed in Chapter 19, "Performance Monitoring Events."

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 18.2.

See also:

- Section 18.2, "Architectural Performance Monitoring"
- Section 18.3, "Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors)"
  - Section 18.3.1, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem"
  - Section 18.3.2, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"
  - Section 18.3.3, "Intel® Xeon® Processor E7 Family Performance Monitoring Facility"
  - Section 18.3.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge"
  - Section 18.3.5, "3rd Generation Intel® Core™ Processor Performance Monitoring Facility"
  - Section 18.3.6, "4th Generation Intel® Core™ Processor Performance Monitoring Facility"
  - Section 18.3.7, "5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility"

- Section 18.3.8, “6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.4, “Performance monitoring (Intel® Xeon™ Phi Processors)”
  - Section 18.4.1, “Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring”
- Section 18.5, “Performance Monitoring (Intel® Atom™ Processors)”
  - Section 18.5.1, “Performance Monitoring (45 nm and 32 nm Intel® Atom™ Processors)”
  - Section 18.5.2, “Performance Monitoring for Silvermont Microarchitecture”
  - Section 18.5.3, “Performance Monitoring for Goldmont Microarchitecture”
  - Section 18.5.4, “Performance Monitoring for Goldmont Plus Microarchitecture”
- Section 18.6, “Performance Monitoring (Legacy Intel Processors)”
  - Section 18.6.1, “Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)”
  - Section 18.6.2, “Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)”
  - Section 18.6.3, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)”
  - Section 18.6.4, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
    - Section 18.6.4.5, “Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
  - Section 18.6.5, “Performance Monitoring and Dual-Core Technology”
  - Section 18.6.6, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache”
  - Section 18.6.7, “Performance Monitoring on L3 and Caching Bus Controller Sub-Systems”
  - Section 18.6.8, “Performance Monitoring (P6 Family Processor)”
  - Section 18.6.9, “Performance Monitoring (Pentium Processors)”
- Section 18.7, “Counting Clocks”
- Section 18.8, “IA32\_PERF\_CAPABILITIES MSR Enumeration”

## 18.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T 7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 1, 2, and 3. Intel processors based on the Skylake, Kaby Lake and Coffee Lake microarchitectures support 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.

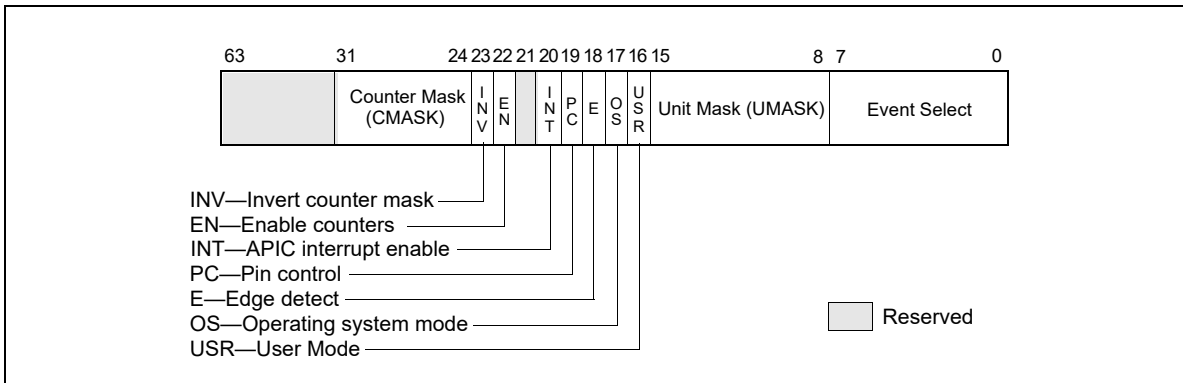


Figure 18-1. Layout of IA32\_PERFEVTSELx MSRs

- **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 precisions on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

## 18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32\_FIXED\_CTR0 through IA32\_FIXED\_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32\_FIXED\_CTR\_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32\_PMCx) via UMASK field in (IA32\_PERFECTSELx), configuring, programming IA32\_FIXED\_CTR\_CTRL for fixed-function PMCs do not require any UMASK.

- Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSR:
  - IA32\_PERF\_GLOBAL\_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32\_PERF\_GLOBAL\_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32\_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32\_DEBUGCTL.Freeze\_LBR\_On\_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32\_DEBUGCTL.Freeze\_PerfMon\_On\_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32\_FIXED\_CTR\_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

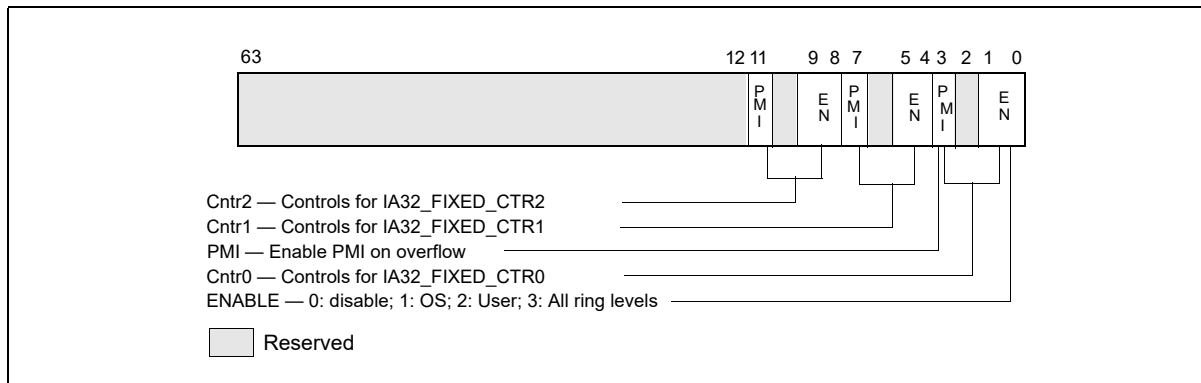


Figure 18-2. Layout of IA32\_FIXED\_CTR\_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32\_PERF\_GLOBAL\_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32\_PERF\_GLOBAL\_CTRL. Each enable bit in IA32\_PERF\_GLOBAL\_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

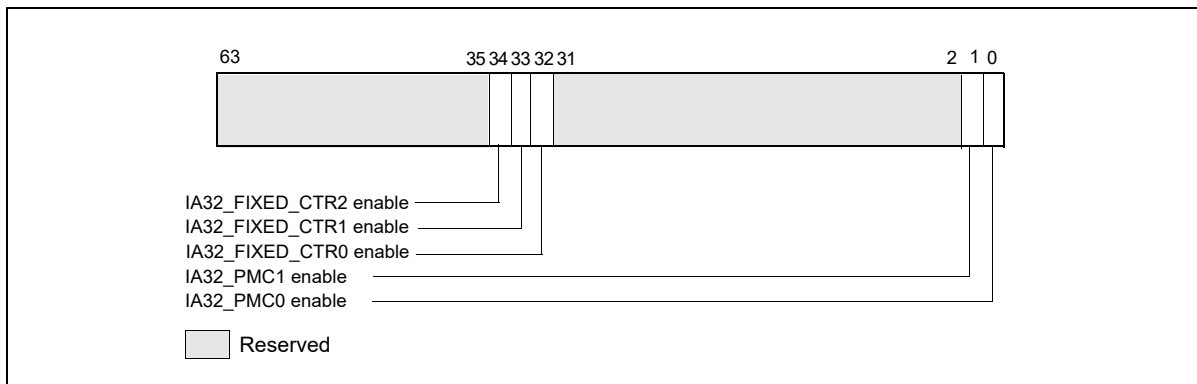


Figure 18-3. Layout of IA32\_PERF\_GLOBAL\_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

**Table 18-2. Association of Fixed-Function Performance Counters with Architectural Performance Events**

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
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 unhalting 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.

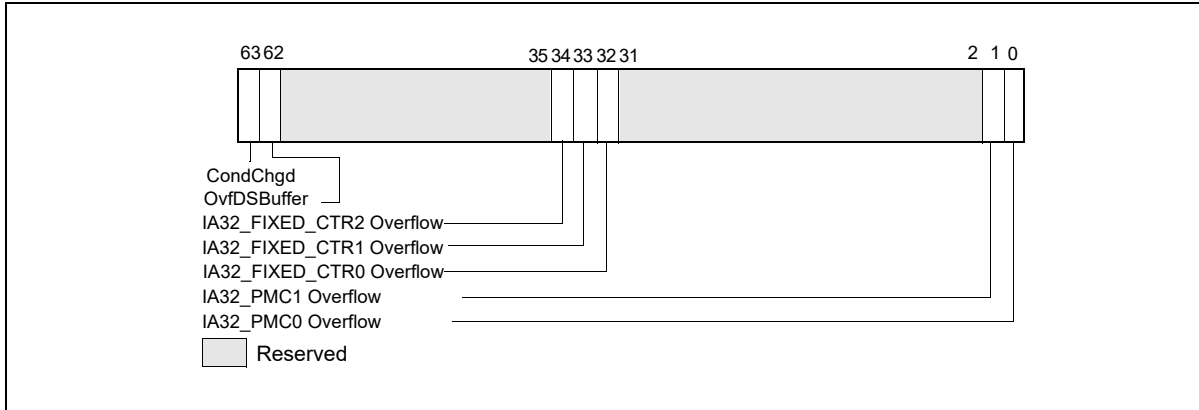


Figure 18-4. Layout of IA32\_PERF\_GLOBAL\_STATUS MSR

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.

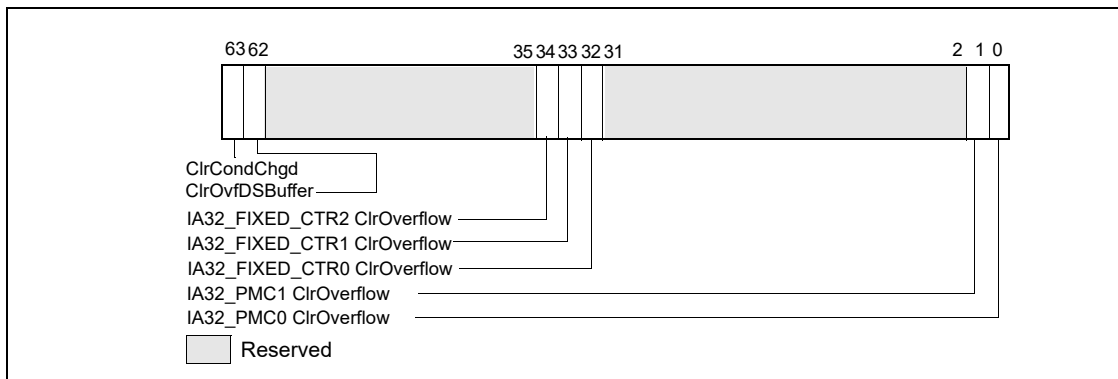


Figure 18-5. Layout of IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR

### 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.

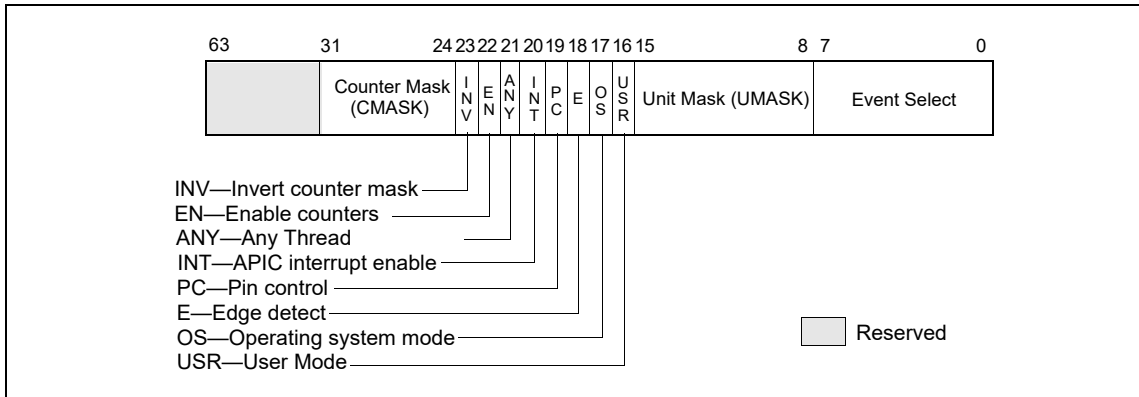


Figure 18-6. Layout of IA32\_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

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.

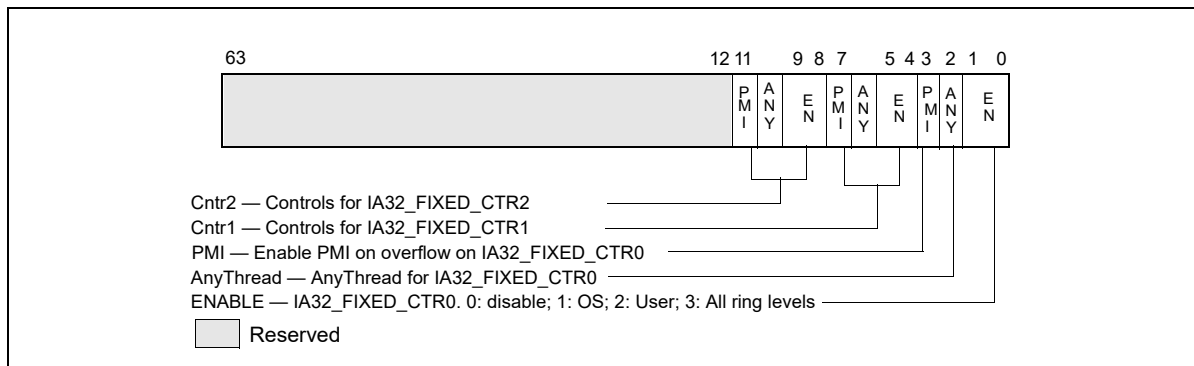
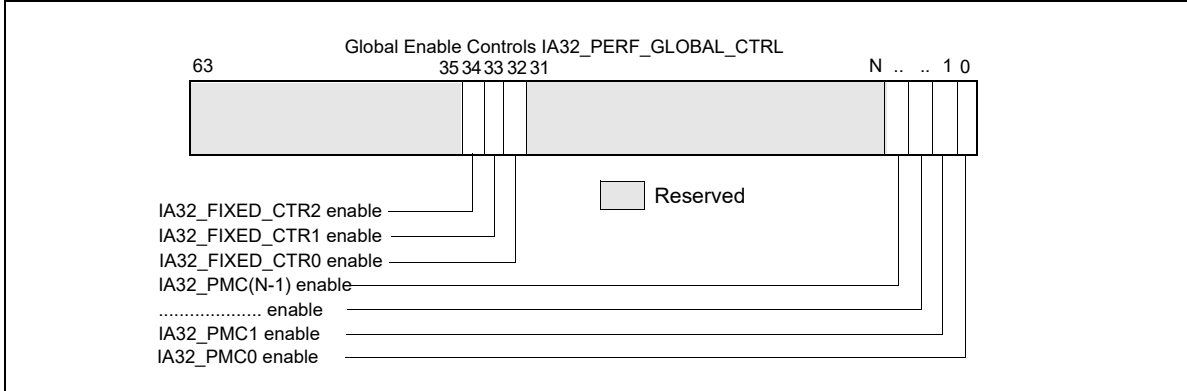


Figure 18-7. IA32\_PERF\_FIXED\_CTR\_CTRL MSR Supporting Architectural Performance Monitoring Version 3

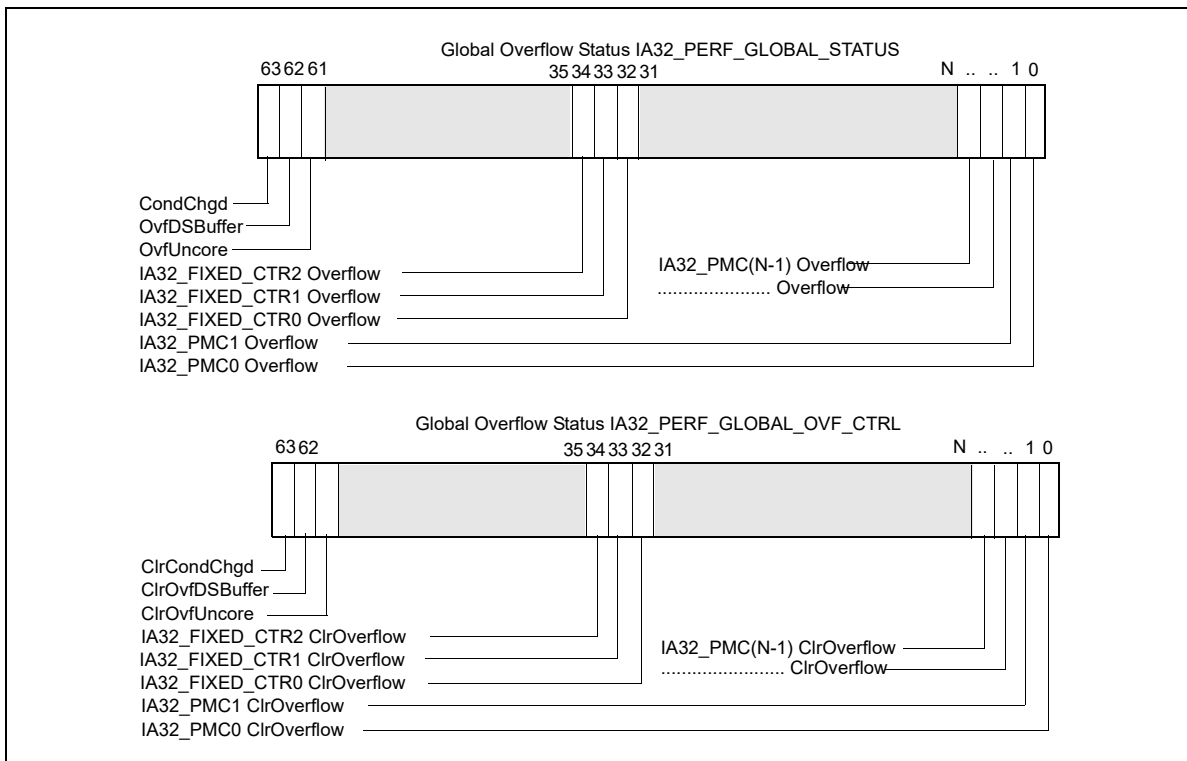
Each control block for a fixed-function performance counter provides a **AnyThread** (bit position 2 + 4\*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread’s CPL with the ENABLE setting of the corresponding control block of IA32\_PERF\_FIXED\_CTR\_CTRL) 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:

- $(\text{IA32\_DEBUGCTL.LBR} \ \& \ (!\text{IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ})) = 1$

- IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ[bit 59]: This bit is set due to the following conditions:
  - IA32\_DEBUGCTL.FREEZE\_PERFMON\_ON\_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

Effectively, the IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- $(IA32\_PERFEVTSELn.EN \ \& \ IA32\_PERF\_GLOBAL\_CTRL.PMCn \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for programmable counter 'n', or
- $(IA32\_PERF\_FIXED\_CTRL.ENi \ \& \ IA32\_PERF\_GLOBAL\_CTRL.FCi \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for fixed counter 'i'

The read-only enable interface IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ provides a more efficient flow for a PMI handler to use IA32\_DEBUGCTL.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).

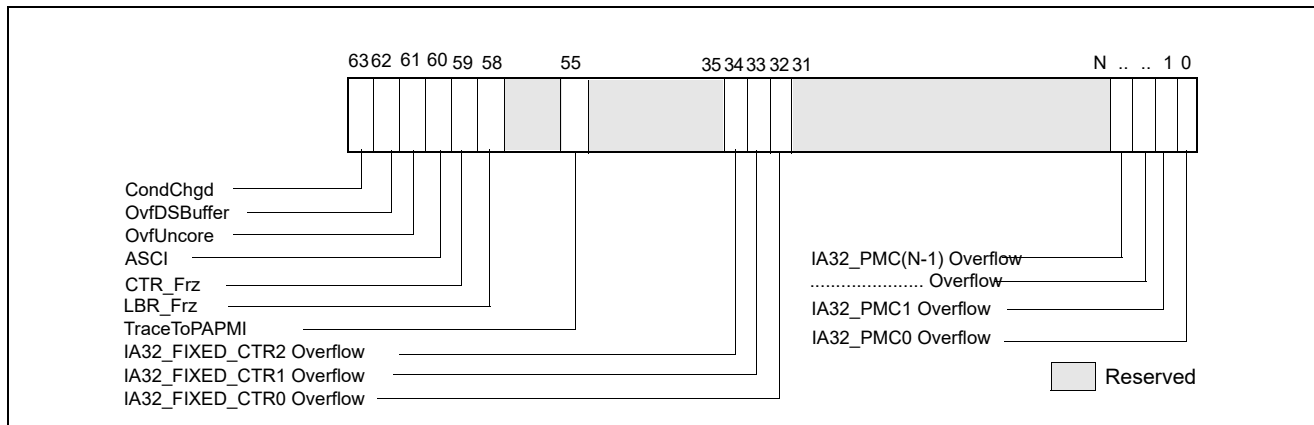


Figure 18-10. IA32\_PERF\_GLOBAL\_STATUS MSR and Architectural Perfmon Version 4

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.

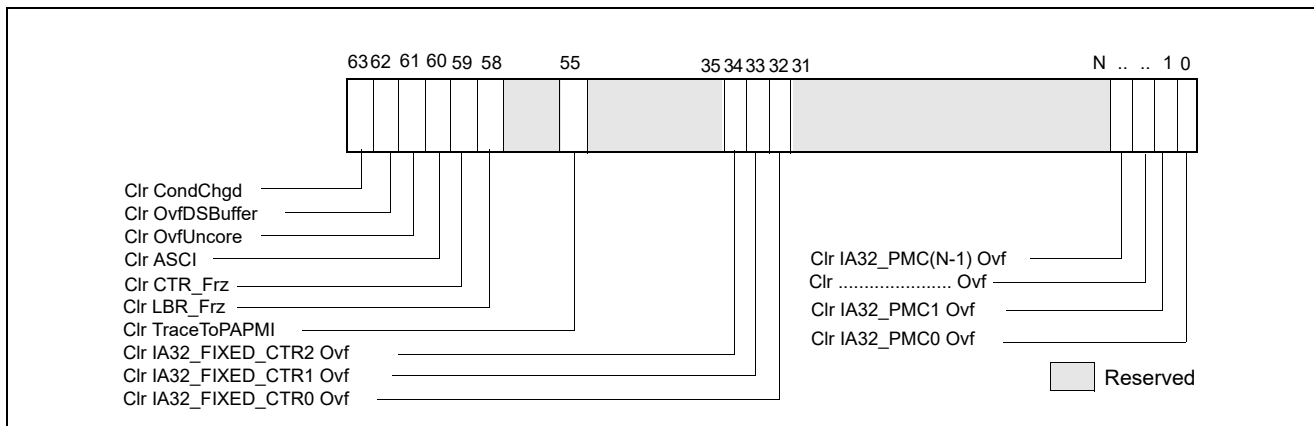


Figure 18-11. IA32\_PERF\_GLOBAL\_STATUS\_RESET MSR and Architectural Perfmon Version 4

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.

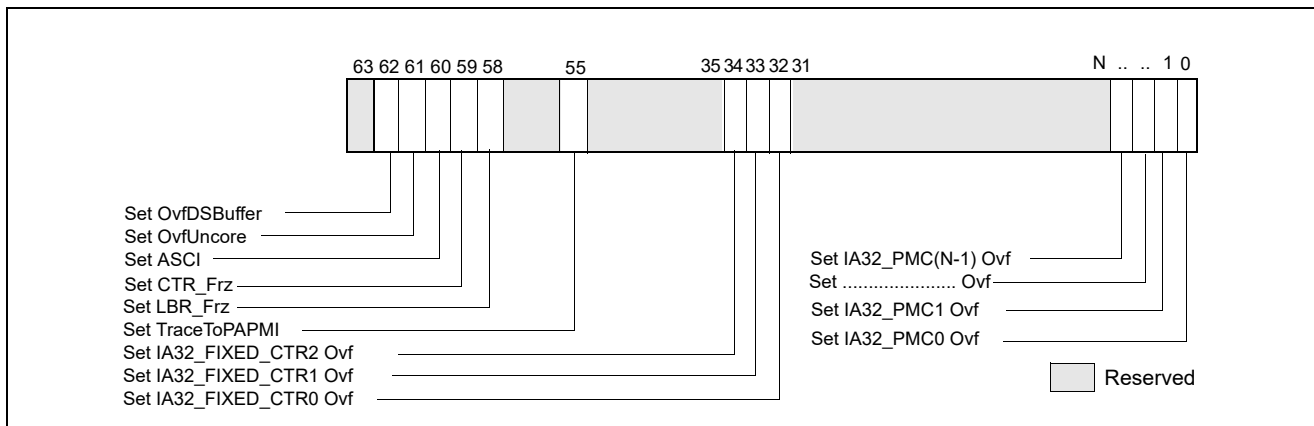


Figure 18-12. IA32\_PERF\_GLOBAL\_STATUS\_SET MSR and Architectural Perfmon Version 4

### 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”<sup>1</sup>, 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.

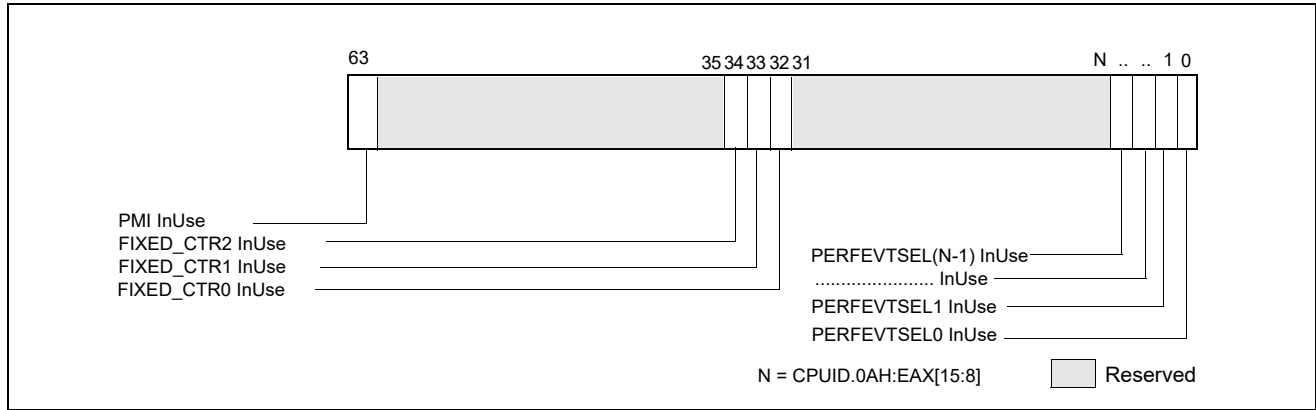


Figure 18-13. IA32\_PERF\_GLOBAL\_INUSE MSR and Architectural Perfmon Version 4

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<sup>2</sup> 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\_PERFVTSELx 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\_PERFVTSELx MSR.

The bit fields within each IA32\_PERFVTSELx 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.

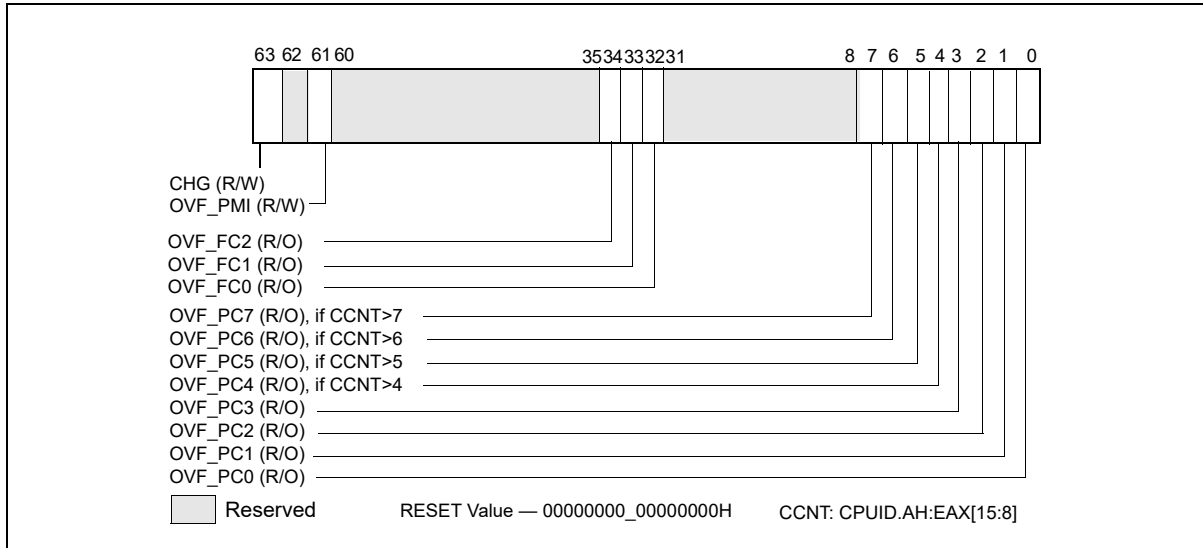


Figure 18-14. IA32\_PERF\_GLOBAL\_STATUS MSR

### 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.

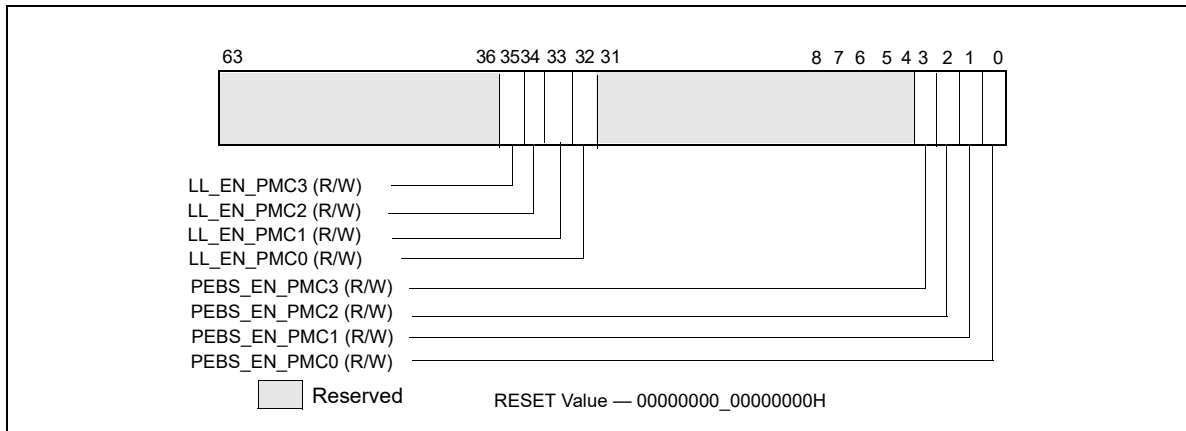


Figure 18-15. Layout of IA32\_PEBS\_ENABLE MSR

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.



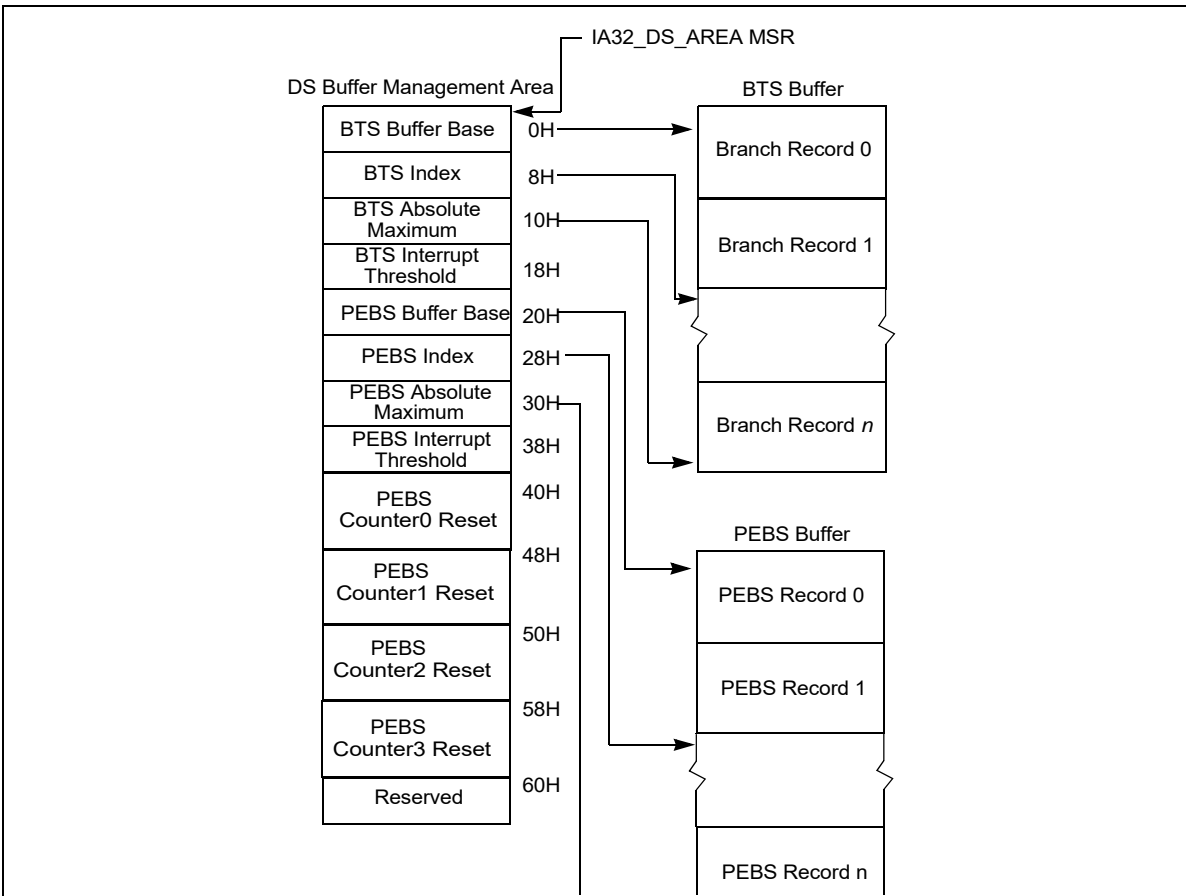


Figure 18-16. PEBS Programming Environment

- 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.

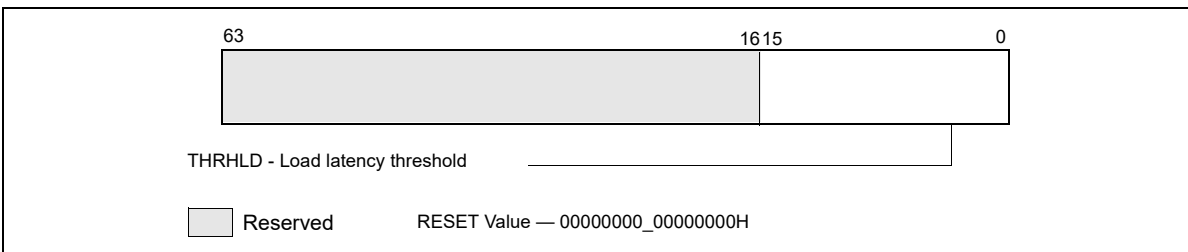
**Table 18-4. Data Source Encoding for Load Latency Record**

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 <sup>1</sup>	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.

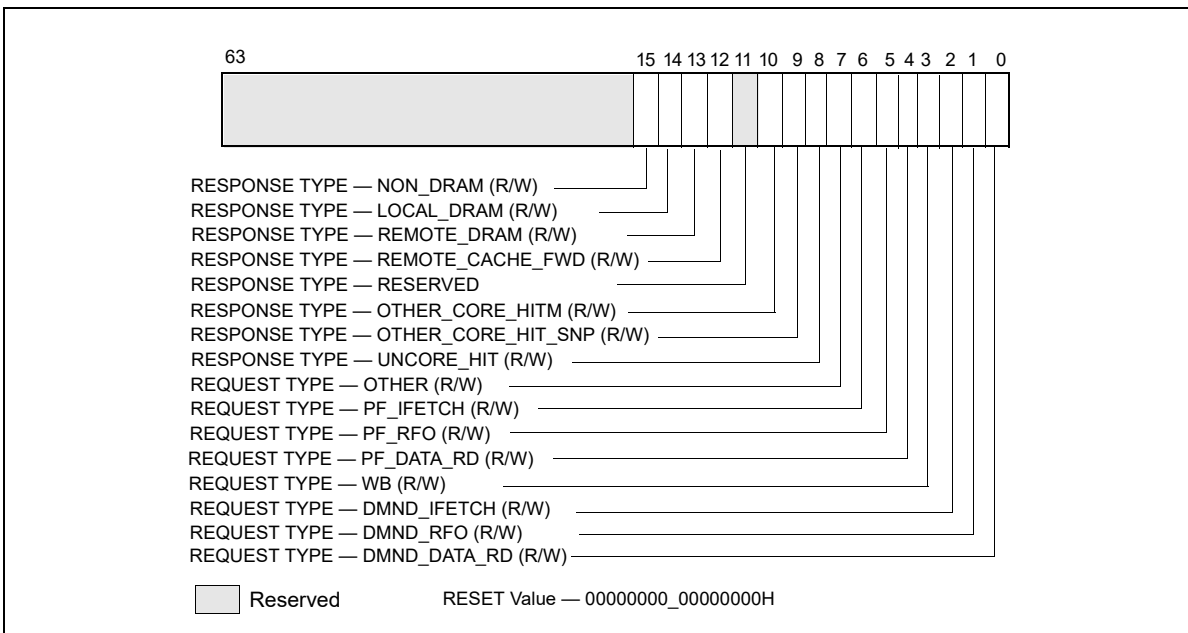
### 18.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

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_HIT_TM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

### 18.3.1.2 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 COH under device number 0 and Function 0.

#### 18.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR\_UNCORE\_PERF\_GLOBAL\_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 18-19 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN\_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR\_UNCORE\_PerfCntr n.
- EN\_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0.

- EN\_PMI\_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32\_DEBUGCTL.Offcore\_PMI\_EN to 1.
- PMI\_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL upon exit from the ISR.

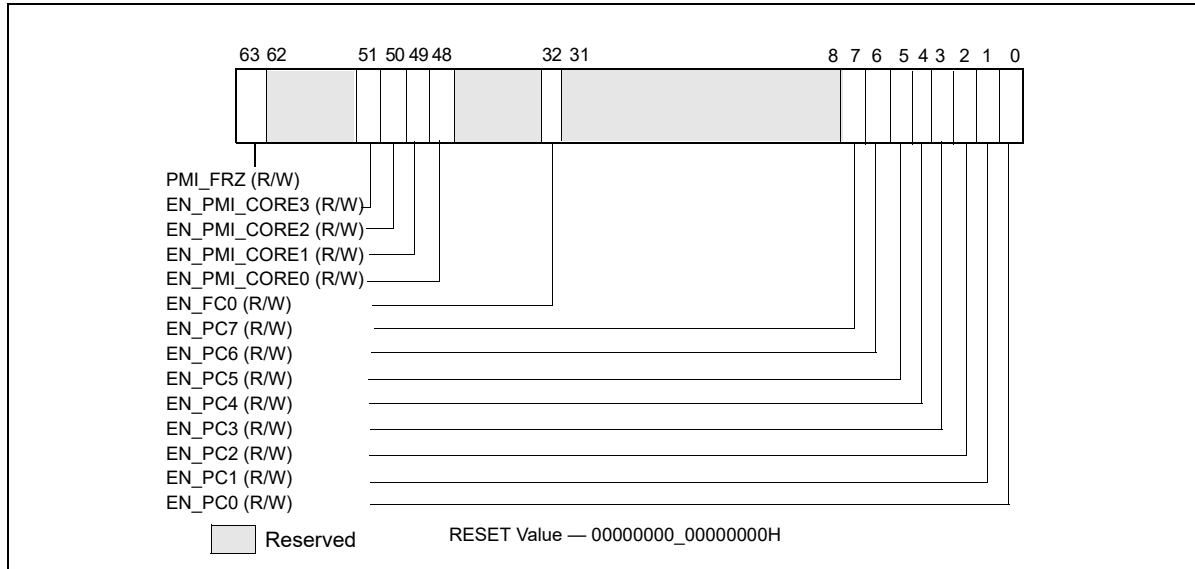


Figure 18-19. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_CTRL MSR

MSR\_UNCORE\_PERF\_GLOBAL\_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 18-20 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_STATUS.

- OVF\_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR\_UNCORE\_PerfCntr n has overflowed.
- OVF\_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0 has overflowed.
- OVF\_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS register has changed state.

MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear the status bits in the UNCORE\_PERF\_GLOBAL\_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

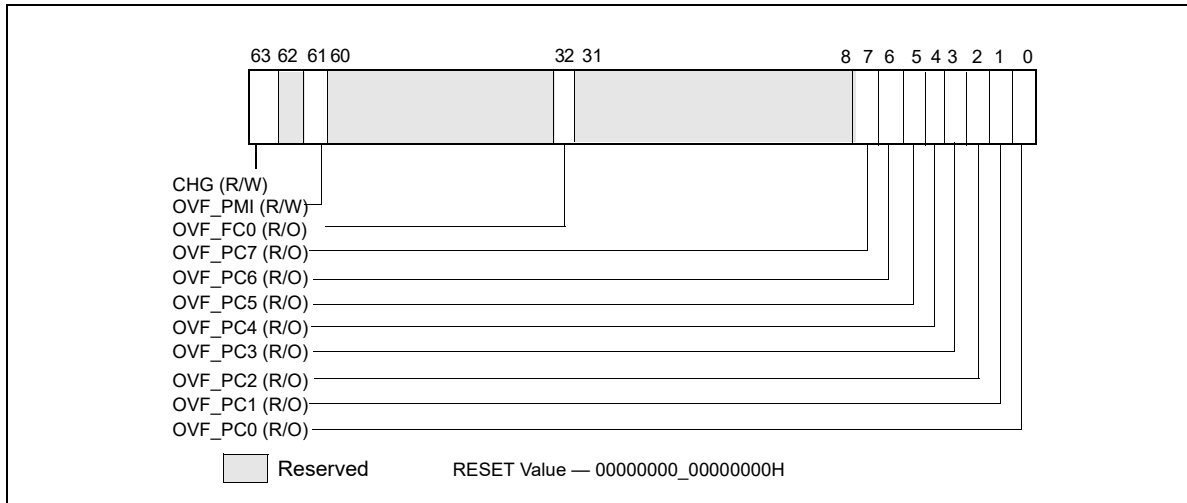


Figure 18-20. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_STATUS MSR

Figure 18-21 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL.

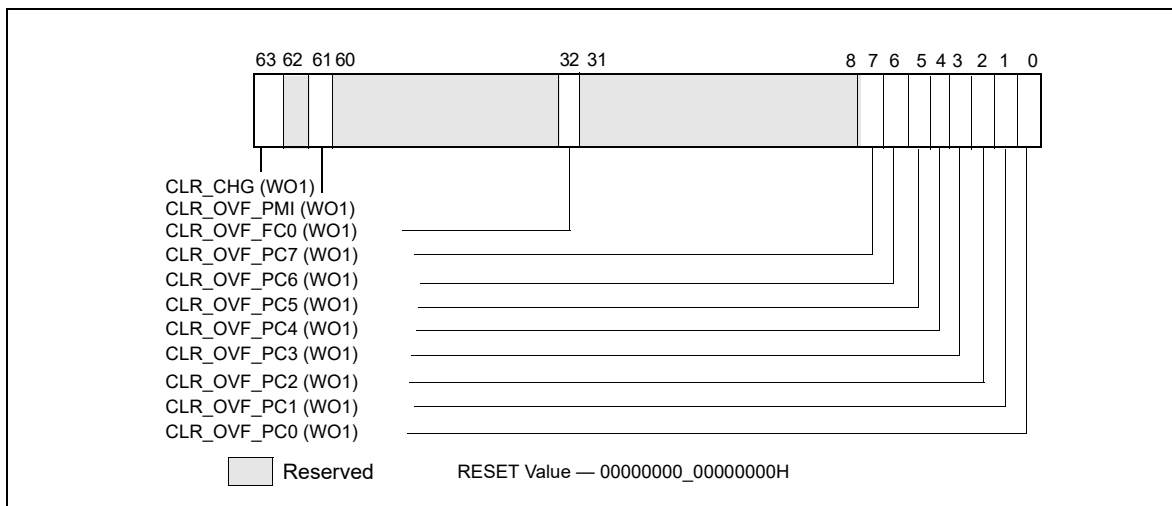


Figure 18-21. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL MSR

- CLR\_OVF\_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR\_UNCORE\_PerfCntr n. Writing a value other than 1 is ignored.
- CLR\_OVF\_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0. Writing a value other than 1 is ignored.
- CLR\_OVF\_PMI (bit 61): Set this bit to clear the OVF\_PMI flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS. Writing a value other than 1 is ignored.
- CLR\_CHG (bit 63): Set this bit to clear the CHG flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS register. Writing a value other than 1 is ignored.

### 18.3.1.2.2 Uncore Performance Event Configuration Facility

MSR\_UNCORE\_PerfEvtSel0 through MSR\_UNCORE\_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR\_UNCORE\_PerfEvtSelx and counting must be enabled using the respective EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL. Figure 18-22 shows the layout of MSR\_UNCORE\_PERFEVTSELx.

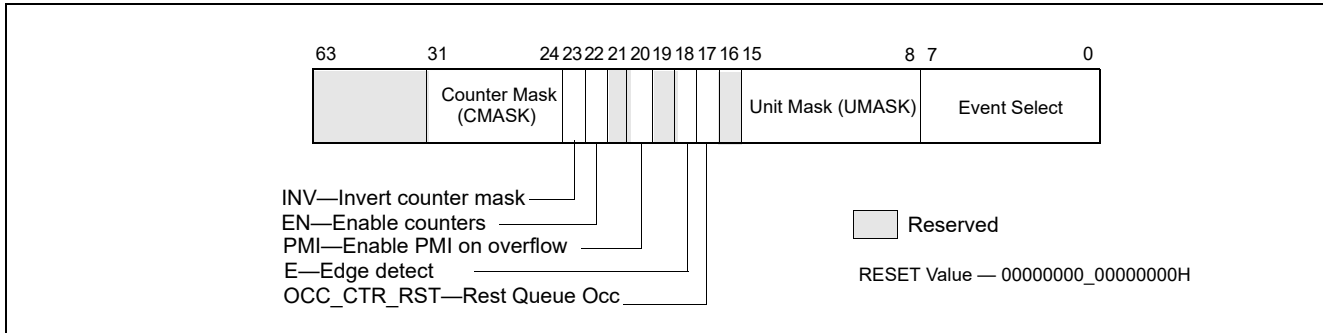


Figure 18-22. Layout of MSR\_UNCORE\_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC\_CTR\_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 18-23 shows the layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL.

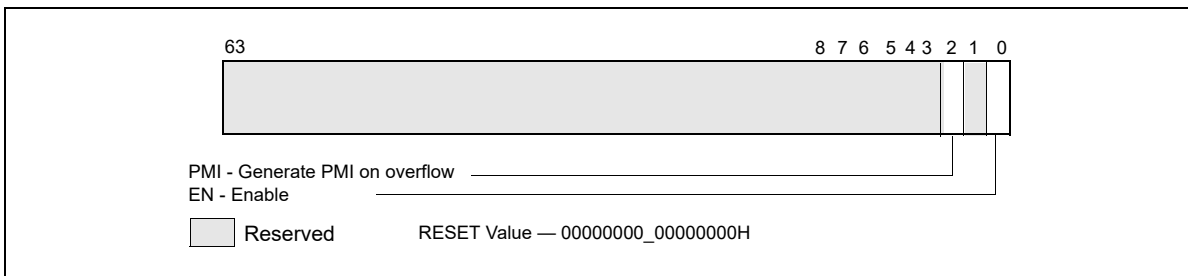


Figure 18-23. Layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL MSR



- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN\_FC0 bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.

Both the general-purpose counters (MSR\_UNCORE\_PerfCnt) and the fixed-function counter (MSR\_UNCORE\_FixedCnt0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

### 18.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR\_UNCORE\_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR\_OPCODE\_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH register. The layout is shown in Figure 18-24.

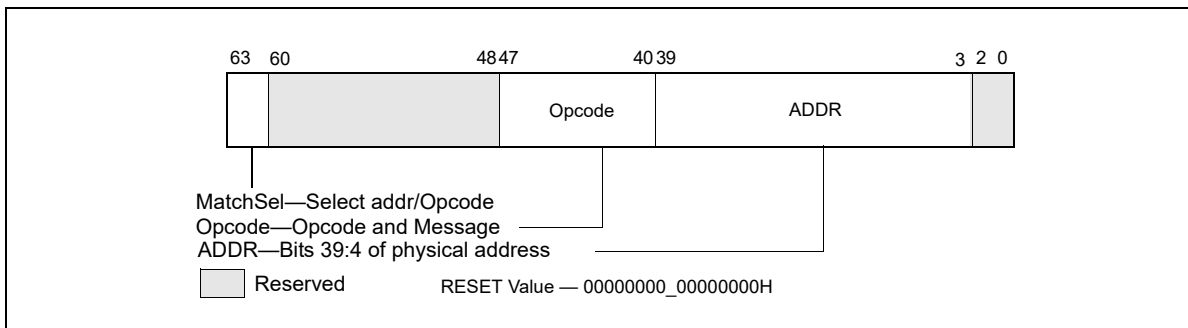


Figure 18-24. Layout of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - Bits 43:40 specify the QPI packet header opcode.
  - Bits 47:44 specify the QPI message classes.

Table 18-7 lists the encodings supported in the opcode field.

**Table 18-7. Opcode Field Encoding for MSR\_UNCORE\_ADDR\_OPCODE\_MATCH**

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr\_opcode match hardware.
  - 100B: Count if only the address field matches.
  - 010B: Count if only the opcode field matches.
  - 110B: Count if either opcode field matches or the address field matches.
  - 001B: Count only if both opcode and address field match.
  - Other encoding are reserved.

### 18.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different. The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 18-25.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

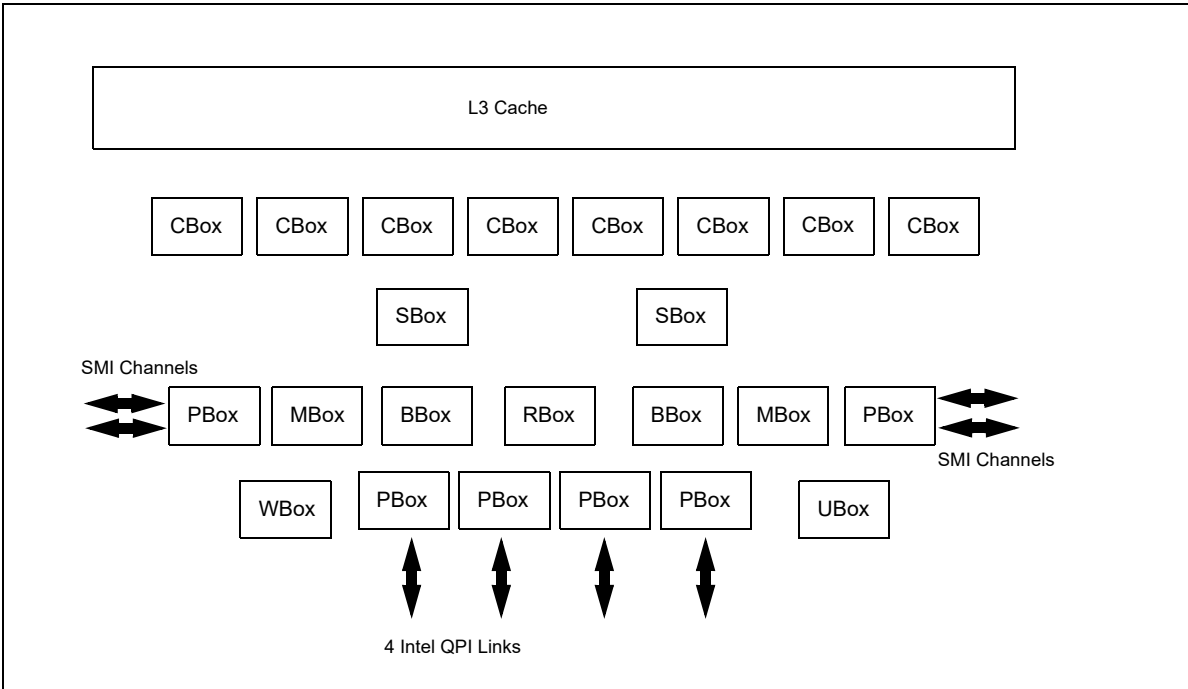


Figure 18-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-8 summarizes the number MSRs for uncore PMU for each box.

Table 18-8. Uncore PMU MSR Summary

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

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the “global control” programming interface, IA32\_PERF\_GLOBAL\_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR\_U\_PMON\_GLOBAL\_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual 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 than one box of the same type, %scope\_function% follows the examples below:

- Multi-counter enabling MSRs: MSR\_U\_PMON\_GLOBAL\_CTL, MSR\_S0\_PMON\_BOX\_CTL, MSR\_C7\_PMON\_BOX\_CTL, etc.
- Multi-counter status MSRs: MSR\_U\_PMON\_GLOBAL\_STATUS, MSR\_S0\_PMON\_BOX\_STATUS, MSR\_C7\_PMON\_BOX\_STATUS, etc.
- Multi-counter overflow control MSRs: MSR\_U\_PMON\_GLOBAL\_OVF\_CTL, MSR\_S0\_PMON\_BOX\_OVF\_CTL, MSR\_C7\_PMON\_BOX\_OVF\_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_CTR, MSR\_S0\_PMON\_CTR0, MSR\_C7\_PMON\_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_EVNT\_SEL, MSR\_S0\_PMON\_EVNT\_SEL0, MSR\_C7\_PMON\_EVNT\_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR\_M0\_PMON\_TIMESTAMP, MSR\_R0\_PMON\_IPERFO\_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<sup>3</sup>. 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**

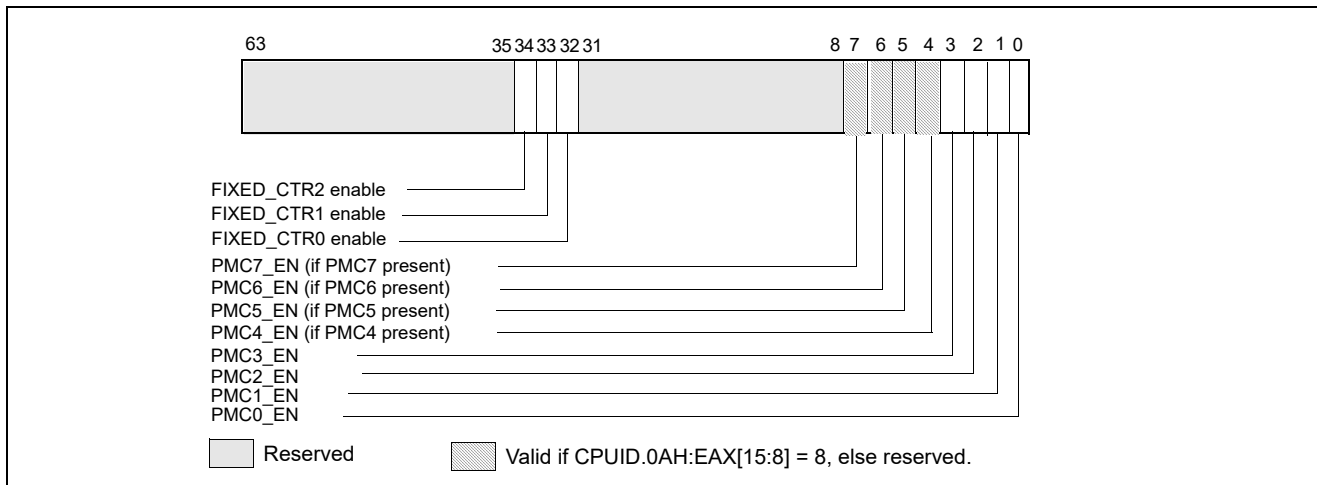
Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	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; <ul style="list-style-type: none"> <li>▪ Data source encoding</li> <li>▪ STLB miss encoding</li> <li>▪ Lock transaction encoding</li> </ul>	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.

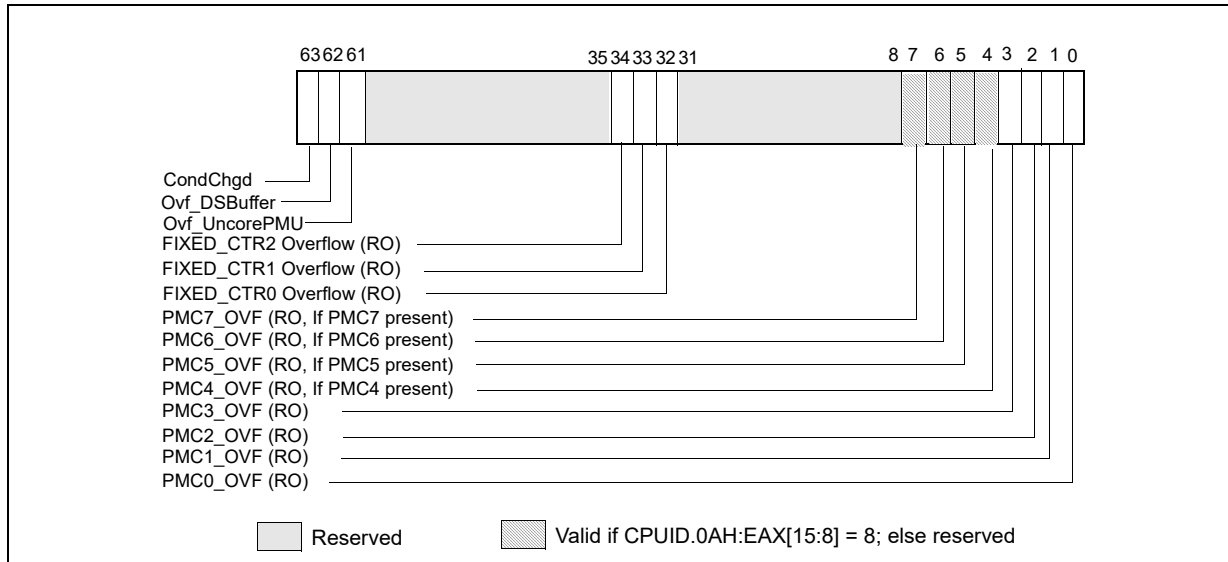


Figure 18-27. IA32\_PERF\_GLOBAL\_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

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.

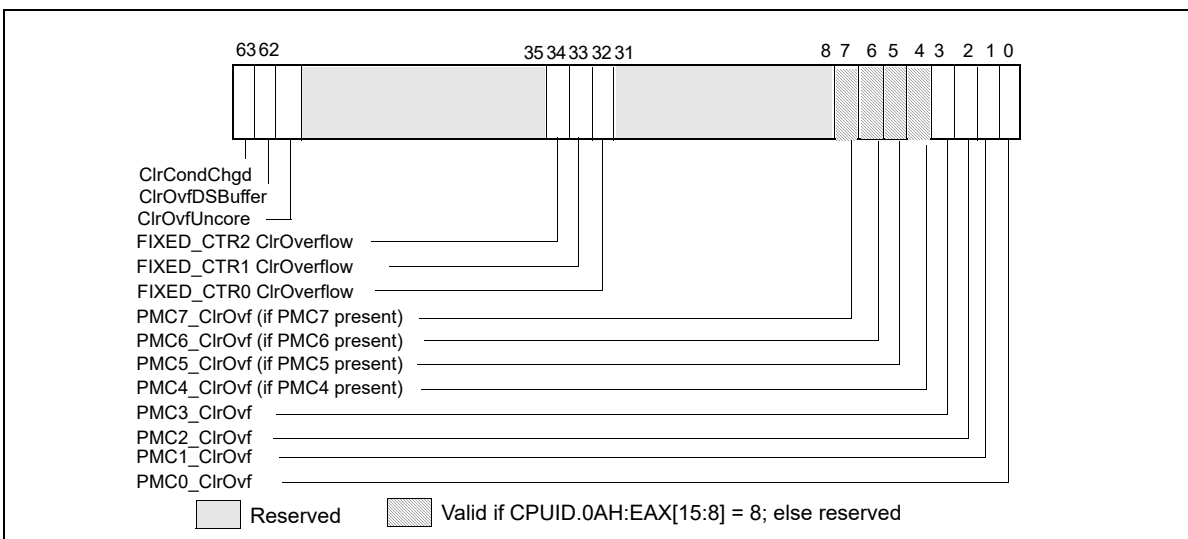


Figure 18-28. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Intel microarchitecture code name Sandy Bridge

### 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	PMCO-PMC3	PMCO-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.

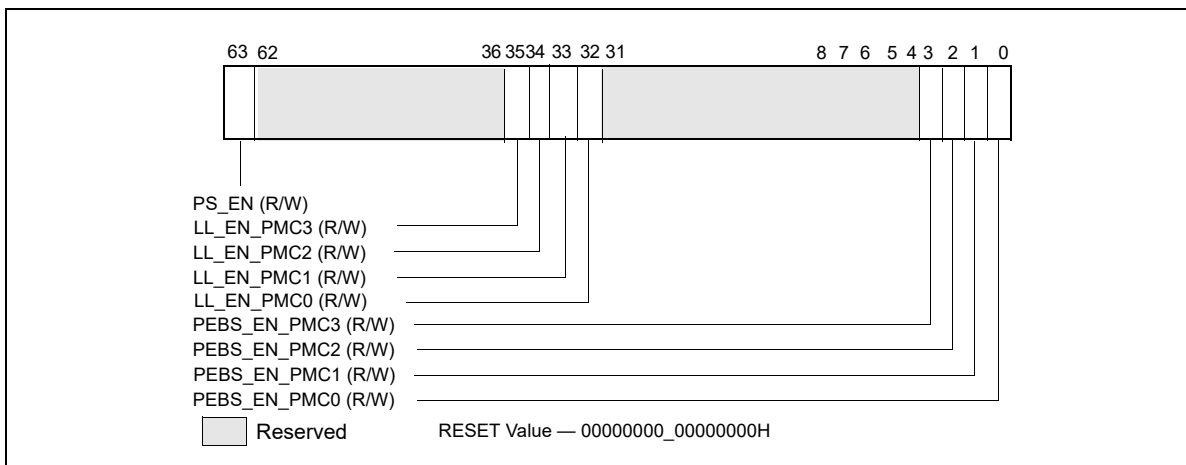


Figure 18-29. Layout of IA32\_PEBS\_ENABLE MSR

#### 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 <sup>1</sup>
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\_RETIRE event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

**Table 18-13. Layout of Data Source Field of Load Latency Record**

Field	Position	Description
Source	3:0	See Table 18-4
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR\_PEBS\_LD\_LAT\_THRESHOLD is the same as shown in Figure 18-17.

#### 18.3.4.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32\_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.

- Program the MEM\_TRANS\_RETIRE.PRECISE\_STORE event in IA32\_PERFVTSEL3. Only counter 3 (IA32\_PMC3) supports collection of precise store information.
- Set IA32\_PEBS\_ENABLE[3] and IA32\_PEBS\_ENABLE[63]. This enables IA32\_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

**Table 18-14. Layout of Precise Store Information In PEBS Record**

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	<p><b>L1D Hit</b> (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.</p> <p><b>STLB Miss</b> (bit 4): The store missed the STLB if set, otherwise the store hit the STLB</p> <p><b>Locked Access</b> (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.</p>
Reserved	A8H	Reserved

**18.3.4.4 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\_RETIRE 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\_RETIRE 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\_RETIRE.ALL precise event, and must use IA32\_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32\_PEBS\_ENABLE set to 1. INST\_RETIRE.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_2A and 06\_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

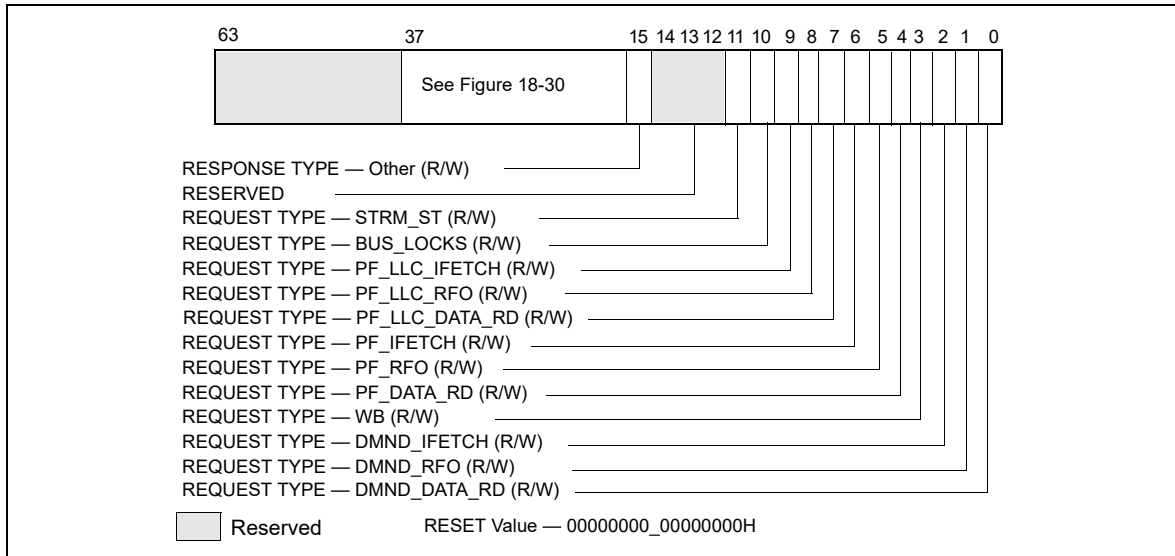
**18.3.4.5 Off-core Response Performance Monitoring**

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Table 18-15 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-15. Off-Core Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMCO-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 are shown in Figure 18-30 and Figure 18-31. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.



**Figure 18-30. Request\_Type Fields for MSR\_OFFCORE\_RSP\_x**

**Table 18-16. MSR\_OFFCORE\_RSP\_x Request\_Type Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	(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.

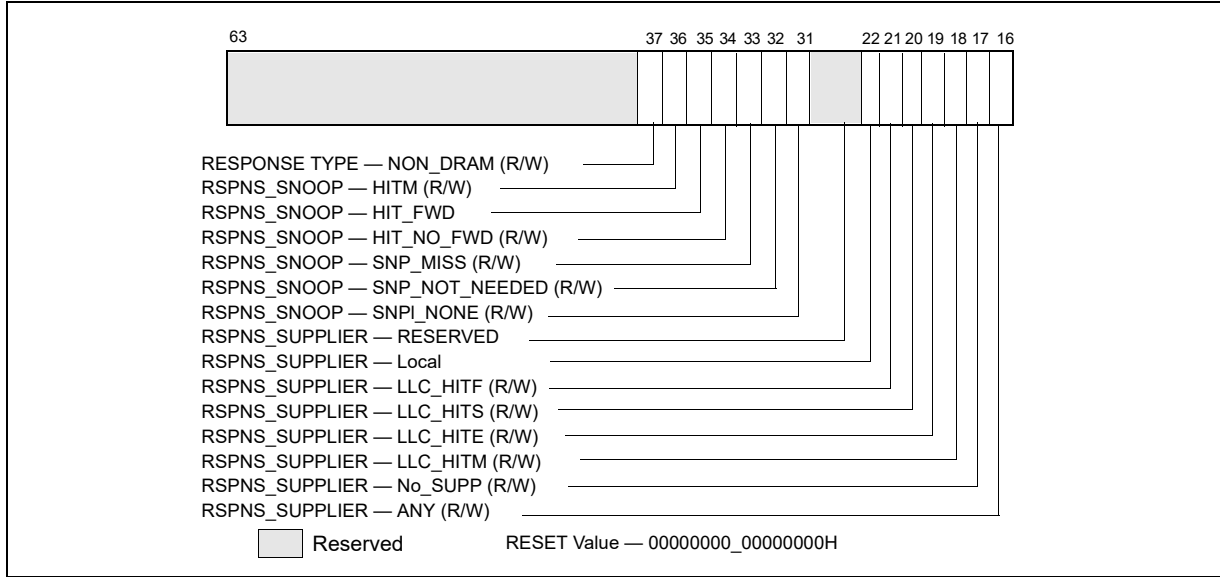


Figure 18-31. Response\_Supplier and Snoop Info Fields for MSR\_OFFCORE\_RSP\_x

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

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)]

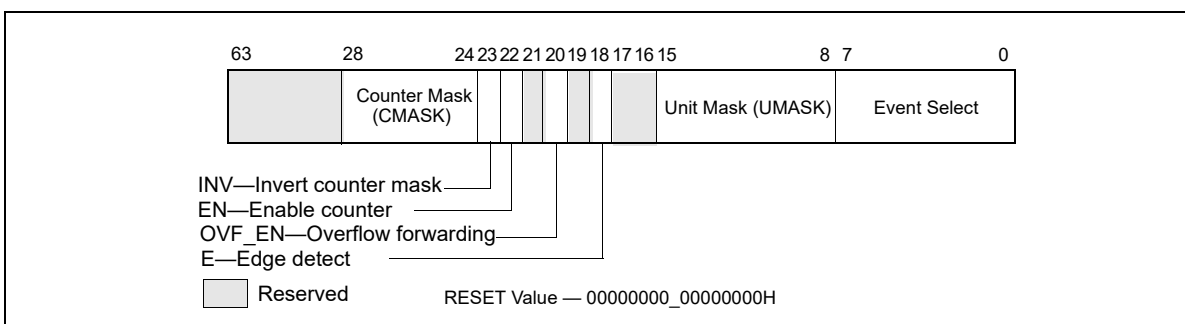
If “ANY” bit is set, the supplier and snoop info bits are ignored.

**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, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

**18.3.4.6 Uncore Performance Monitoring Facilities In Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series**

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 18.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 18-32.



**Figure 18-32. Layout of Uncore PERFVTSSEL MSR for a C-Box Unit or the ARB Unit**

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event\_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see Table 19-18.
- 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).

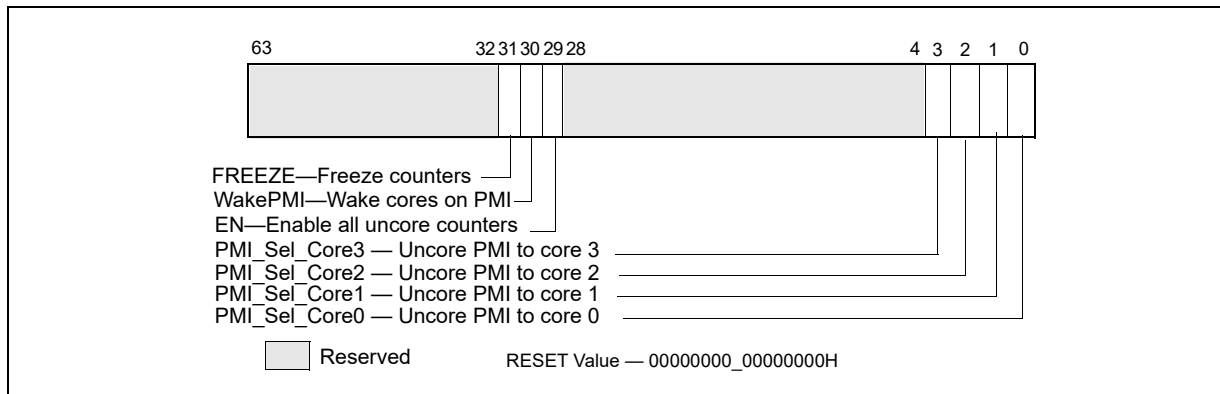


Figure 18-33. Layout of MSR\_UNC\_PERF\_GLOBAL\_CTRL MSR for Uncore



Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSR for uncore PMU for each box.

**Table 18-19. Uncore PMU MSR Summary**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, see Table 2-20 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, there 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	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	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	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (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</li> <li>▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 18-25.</li> </ul>
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.

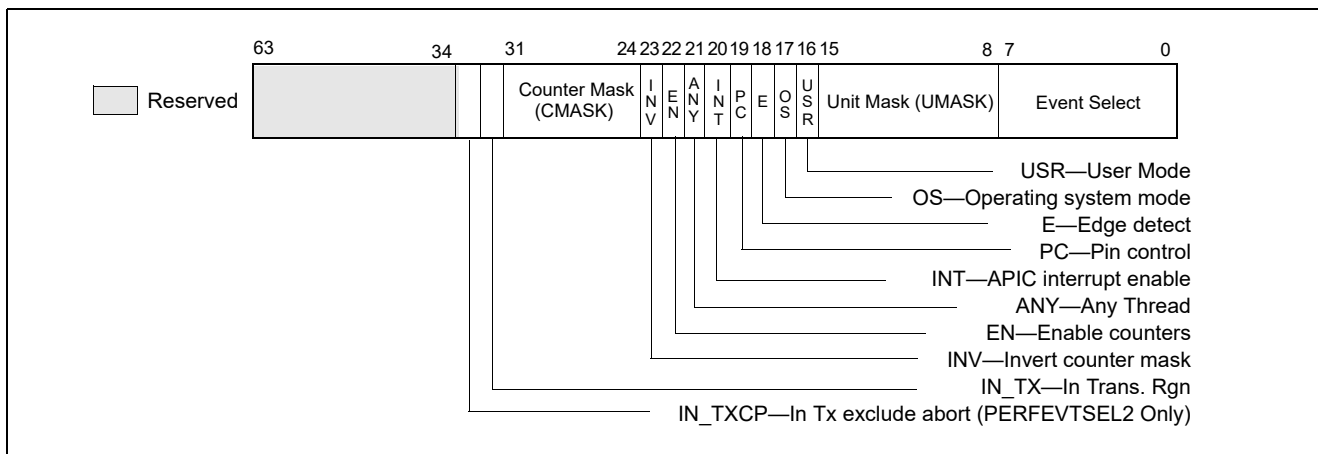
If a processor supports Intel TSX, the core PMU enhances its IA32\_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32\_PERFEVTSELx is shown in Figure 18-34. The two additional bit fields are:



- **IN\_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN\_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32\_PERFEVTSEL2.

When the IA32\_PERFEVTSELx MSR is programmed with both IN\_TX=0 and IN\_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN\_TXCP bit. see Table 2-28.



**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, see Table 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

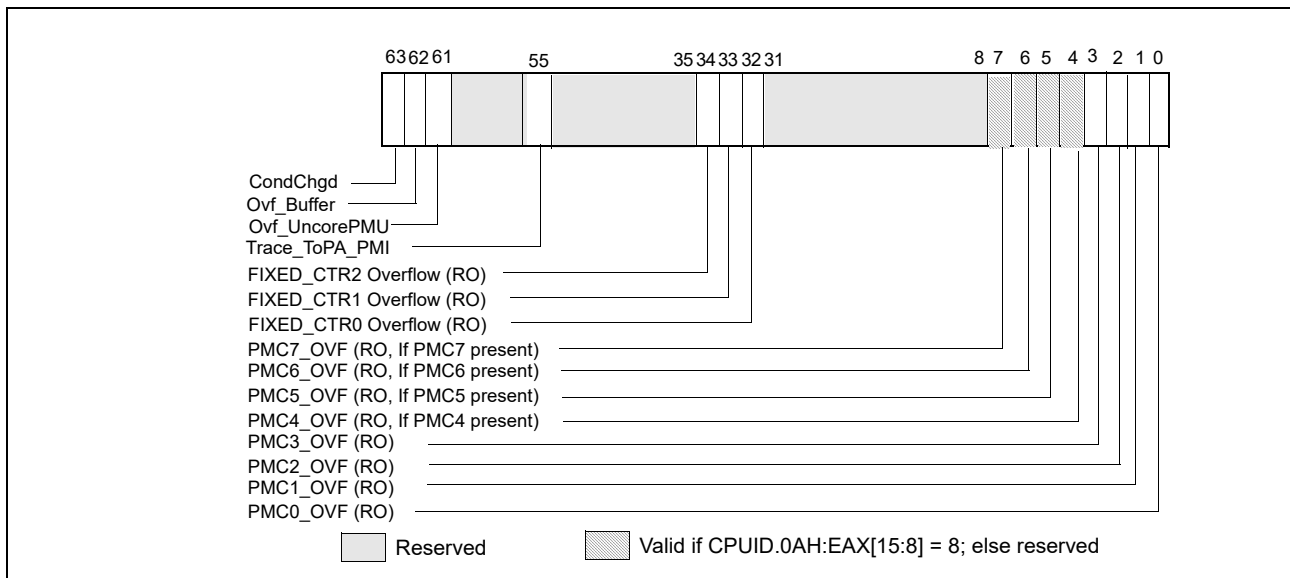
Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in “Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-32.

### 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.

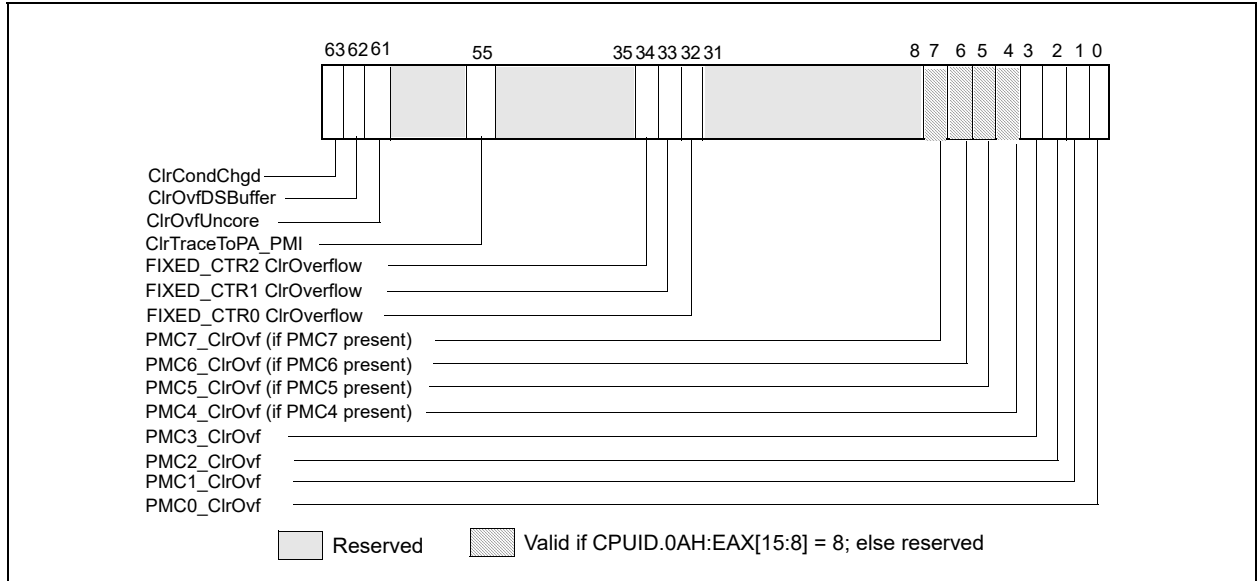


Figure 18-36. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, “Performance Monitoring Events”.

### 18.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The 8th generation Intel® Core™ processor is based on the Coffee Lake microarchitecture. For these microarchitectures, the core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU’s capability is similar to those described in Section 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 42, “Enclave Code Debug and Profiling”.

Table 18-33. Core PMU Comparison

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee Lake	Intel® Microarchitecture Code Name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	
# 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	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>▪ Freeze_on_LBR with streamlined semantics.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>▪ Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> <li>▪ ToPA buffer overflow</li> <li>▪ CTR_Frz, LBR_Frz, ASCI</li> </ul>	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> <li>▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>▪ CTR_Frz</li> <li>▪ LBR_Frz</li> </ul>	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.3.8.1.4.	No	
LBR Record Format Encoding	000101b	000100b	Section 17.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 17.12
LBR Timing	Yes	No	Section 17.12.1
Call Stack Profiling	Yes, see Section 17.11	Yes, see Section 17.11	Use LBR facility
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 18.3.6.5.	See Section 18.3.6.5.	

### 18.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-34.

**Table 18-34. PEBS Facility Comparison**

Box	Intel® Microarchitecture Code Name Skylake, Kaby Lake and Coffee Lake	Intel® Microarchitecture Code Name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMCO-PMC3	PMCO-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 COH.	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

**NOTES**

Precise events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

**18.3.8.1.1 PEBS Data Format**

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32\_PERF\_CAPABILITIES[11:8]. The lay out is shown in Table 18-35. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 18-35. PEBS Record Format for 6th Generation, 7th Generation and 8th Generation Intel Core Processor Families**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-24.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and data address profiling (Section 18.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32\_PERF\_GLOBAL\_STATUS may be useful to resolve the situations when more than one of IA32\_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

### 18.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake and Coffee Lake microarchitectures is shown in Table 18-36.

**Table 18-36. Precise Events for the Skylake, Kaby Lake and Coffee Lake Microarchitectures**

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST <sup>1</sup>	01H
		ALL_CYCLES <sup>2</sup>	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 <sup>3</sup> >	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED <sup>2</sup>	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED <sup>4</sup>	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 <sup>2</sup>	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 generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.



**Table 18-37. Layout of Data Linear Address Information In PEBS Record**

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (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_RETIRE.ALL (if store is tagged), MEM_INST_RETIRE.STLB_MISS_STORES, MEM_INST_RETIRE.ALL_STORES, MEM_INST_RETIRE.SPLIT_STORES.</li> <li>▪ Other bits are zero.</li> </ul>
Reserved	A8H	Always zero.

#### 18.3.8.1.4 PEBS Facility for Front End Events

In the 6th generation, 7th generation and 8th generation Intel Core processors, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The 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\_RETIRE" 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\_RETIRE" 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.
L11_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss <sup>1</sup> . Additional requests to the same cache line as an in-flight L11 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	<p>An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles.</p> <p>The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.</p>

#### NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR\_PEBS\_FRONTEND is given in Table 18-39.

**Table 18-39. MSR\_PEBS\_FRONTEND Layout**

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 18-38.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

### 18.3.8.1.5 FRONTEND\_RETIRED

The FRONTEND\_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) front-end events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a front-end (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

### 18.3.8.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-40.
- Supplier information (bits 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, Kaby Lake and Coffee 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 generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06\_4EH, 06\_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06\_8EH, 06\_9EH).

**Table 18-41. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signatures 06\_4EH, 06\_5EH and 06\_8EH, 06\_9EH)**

Subtype	Bit Name	Offset	Description
Common	Any	16	(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, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

#### 18.3.8.2.1 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_HOPO_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)

### NOTE

This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

### 18.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel® Atom™ processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

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.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.
		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.

**Table 18-48. Bit fields of the MSR\_OFFCORE\_RESP [0, 1] Registers (Contd.)**

Main	Sub-field	Bit	Name	Description
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMCO. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMCO. This bit is reserved for OFFCORE_RESP_1 event.

### 18.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR\_OFFCORE\_RSP0.[bit 38] with the choice of request type specified in MSR\_OFFCORE\_RSP0.[bit 15:0].

Refer to Section 18.5.2.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR\_OFFCORE\_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

## 18.5 PERFORMANCE MONITORING (INTEL® ATOM™ PROCESSORS)

### 18.5.1 Performance Monitoring (45 nm and 32 nm Intel® Atom™ Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 18.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2).

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

**Table 18-49. PEBS Performance Events for the Silvermont Microarchitecture (Contd.)**

Event Name	Event Select	Sub-event	UMask
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
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.

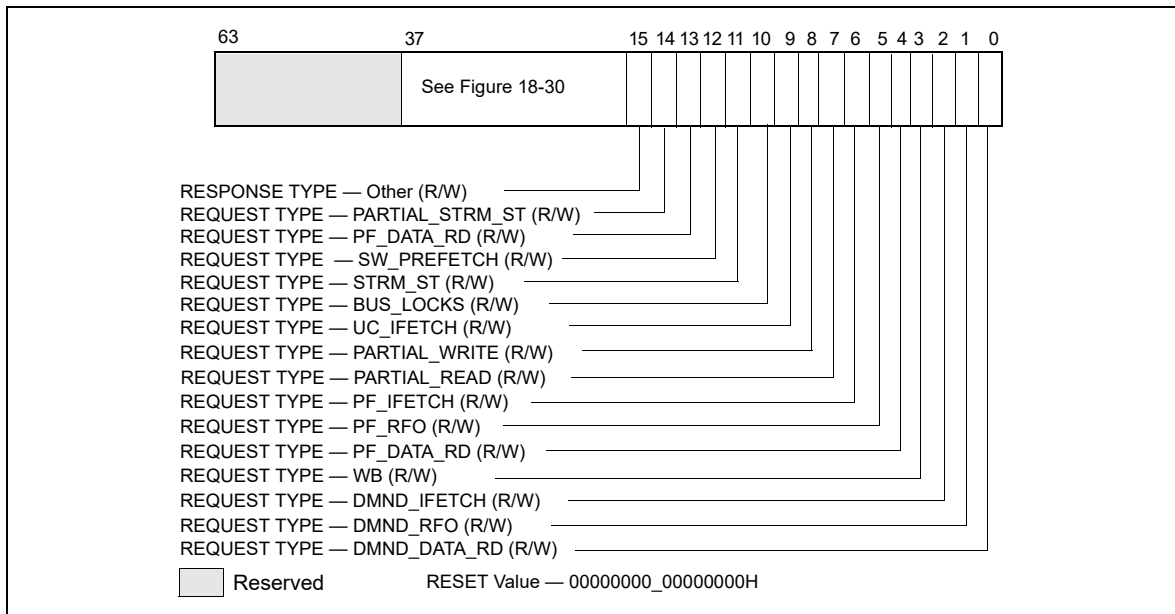
In the Silvermont microarchitecture, each MSR\_OFFCORE\_RSPx is shared by two processor cores.

**Table 18-51. OffCore Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are shown in Figure 18-37 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.



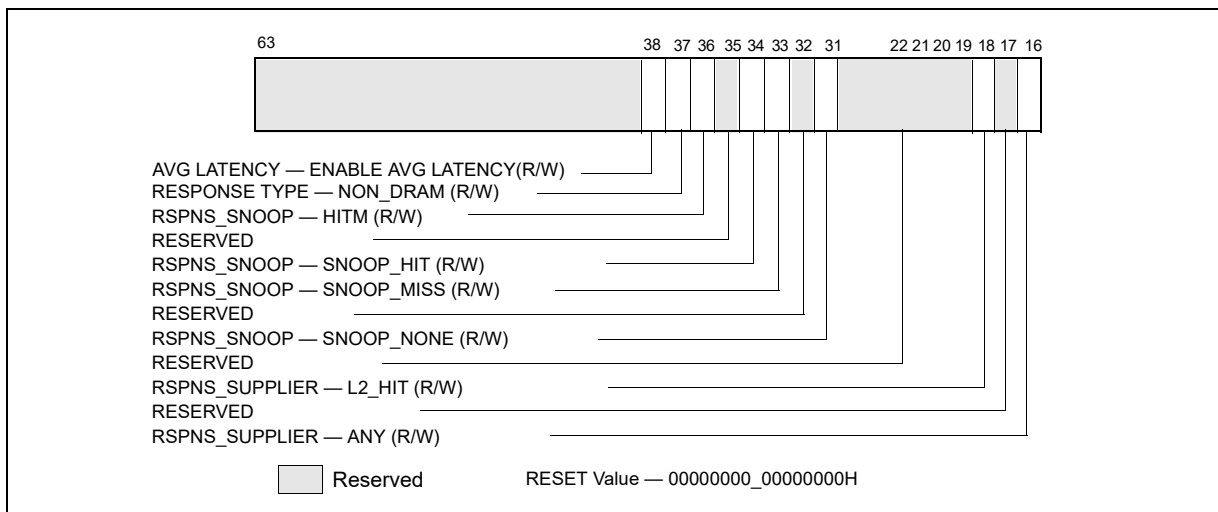
**Figure 18-37. Request\_Type Fields for MSR\_OFFCORE\_RSPx**

**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).

**Table 18-52. MSR\_OFFCORE\_RSPx Request\_Type Field Definition (Contd.)**

Bit Name	Offset	Description
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\_PERFECTSELx MSRs. Count the weighted cycles via MSR\_OFFCORE\_RSP0 by programming a request type in MSR\_OFFCORE\_RSP0.[15:0] and setting MSR\_OFFCORE\_RSP0.OUTSTANDING[38] to 1, while setting the remaining bits to 0. Count the number of requests via MSR\_OFFCORE\_RSP1 by programming the same request type from MSR\_OFFCORE\_RSP0 into MSR\_OFFCORE\_RSP1[bit 15:0], and setting MSR\_OFFCORE\_RSP1.ANY\_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32\_PMCx register that counted weight cycles by the register that counted requests.

## 18.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 18.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The bit fields (except bit 21) within each IA32\_PERFECTSELx 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\_PERFECTSELx 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	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>▪ Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> <li>▪ ToPA buffer overflow</li> <li>▪ CTR_Frz, LBR_Frz</li> </ul>	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>▪ CTR_Frz,</li> <li>▪ LBR_Frz</li> </ul>	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.
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 overflows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE\_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32\_PERF\_GLOBAL\_STATUS\_SET.

**Table 18-56. 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	COH	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

**Table 18-56. Precise Events Supported by the Goldmont Microarchitecture (Contd.)**

Event Name	Event Select	Sub-event	UMask
		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\_RETIRE, except for UOPS\_RETIRE. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

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\_Threshold, 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.
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_0 R_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_CO RE_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_C ORE	36	(R/W) Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	(R/W) Target was a non-DRAM system address. This includes MMIO transactions.

**Table 18-59. MSR\_OFFCORE\_RSPx For L2 Miss and Outstanding Requests (Contd.)**

Subtype	Bit Name	Offset	Description
Outstanding requests <sup>1</sup>	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).

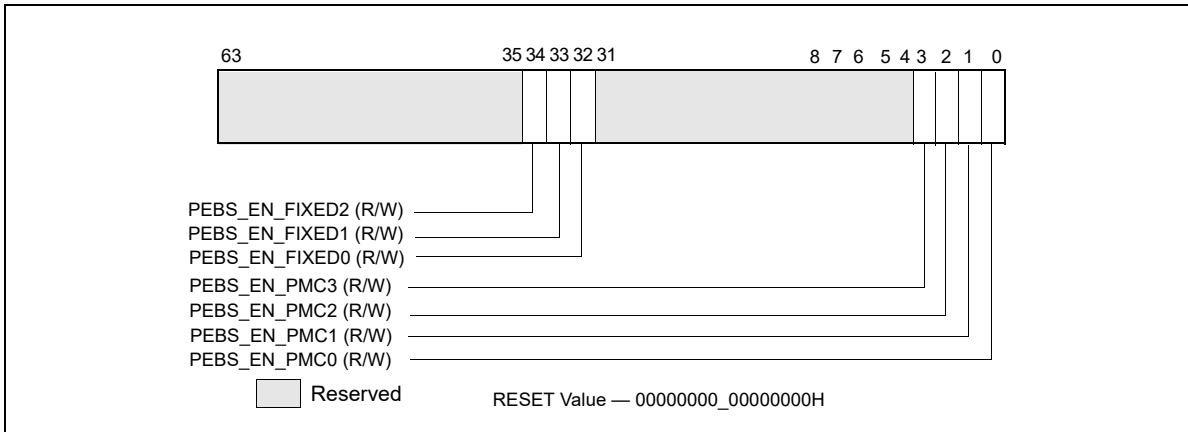


Figure 18-39. Layout of IA32\_PEBS\_ENABLE MSR

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 unhalting core cycles and is a non-precise event. IA32\_FIXED\_CTR2 counts unhalting 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).

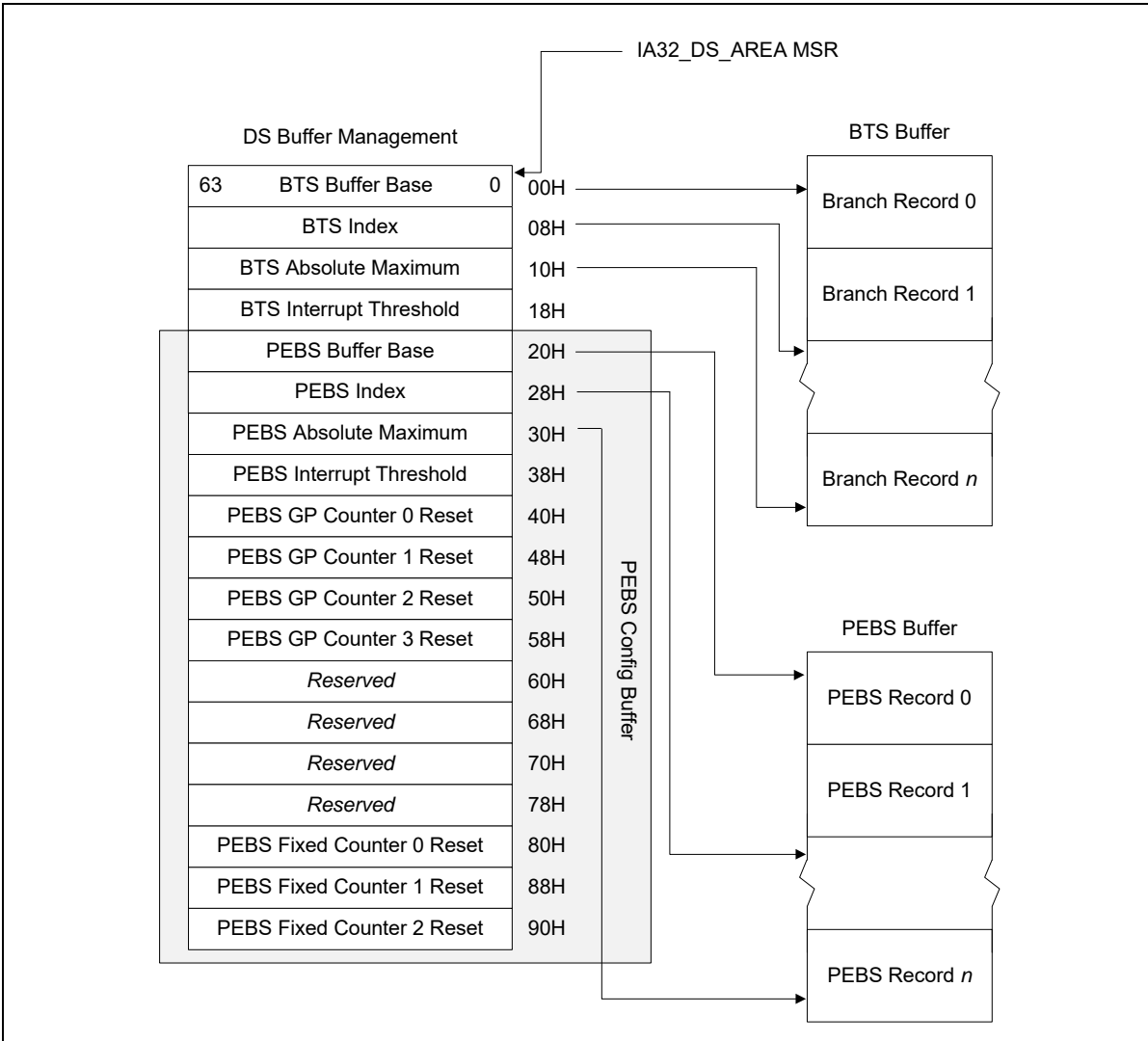


Figure 18-40. PEBS Programming Environment

### 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 field 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.

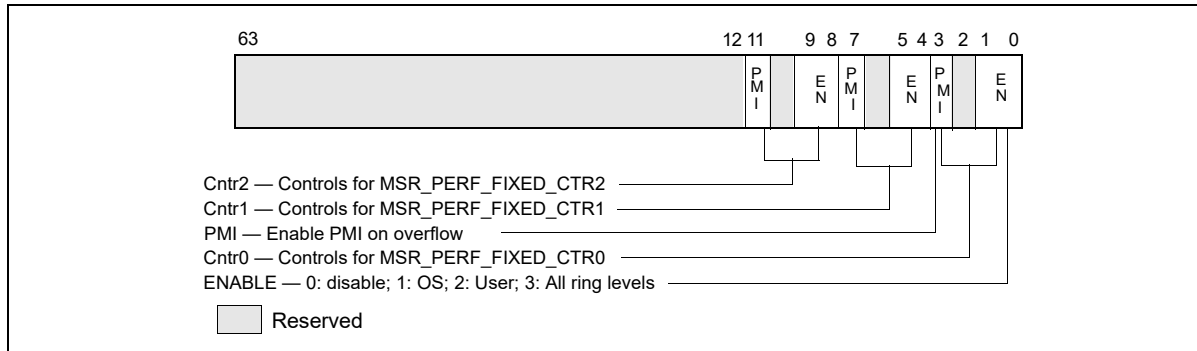


Figure 18-41. Layout of MSR\_PERF\_FIXED\_CTRL MSR

- **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 MSR:

- MSR\_PERF\_GLOBAL\_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (MSR\_PERF\_FIXED\_CTRLx) 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\_CTRLx) 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\_CTRLx) 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\_CTRL\_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.

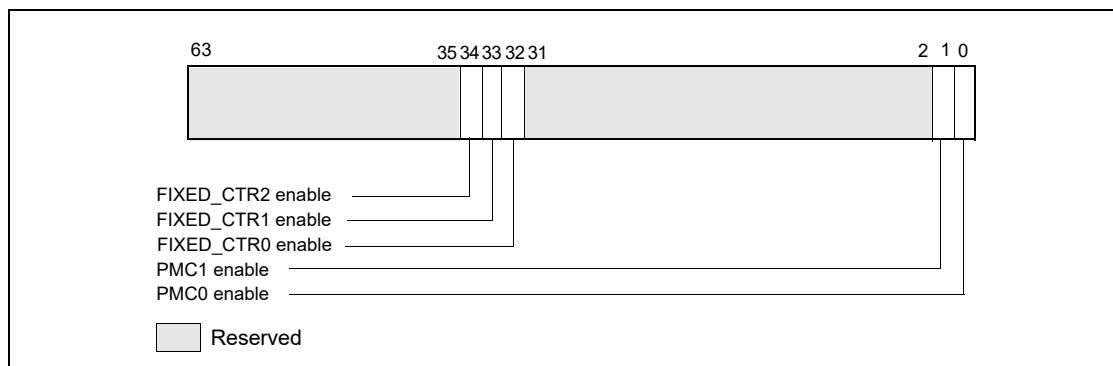


Figure 18-42. Layout of MSR\_PERF\_GLOBAL\_CTRL MSR

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.

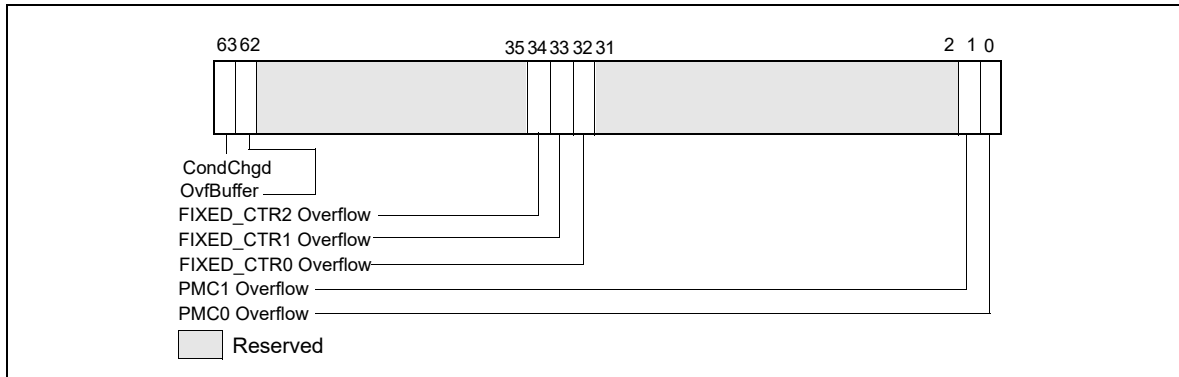


Figure 18-43. Layout of MSR\_PERF\_GLOBAL\_STATUS MSR

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.

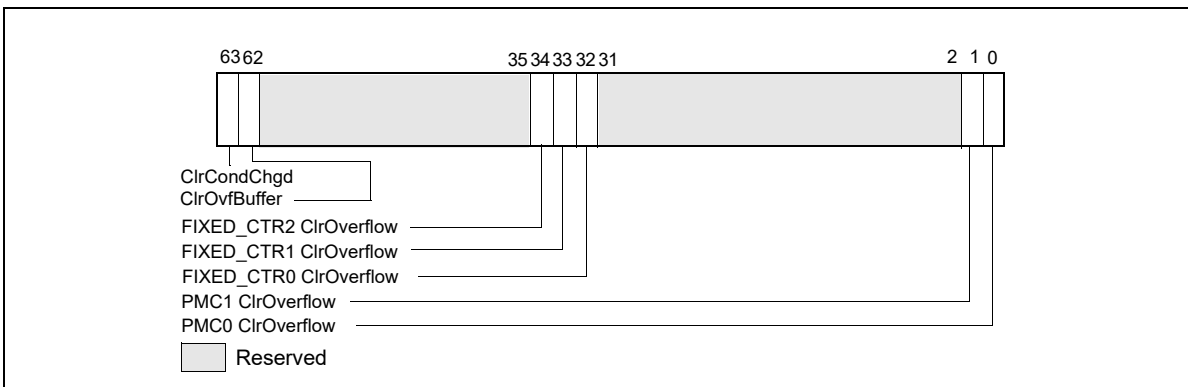


Figure 18-44. Layout of MSR\_PERF\_GLOBAL\_OVF\_CTL MSR

### 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<sup>®</sup> microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 18.6.3.6, "At-Retirement Counting"), but is limited to IA32\_PMC0. See Table 18-67 for a list of events available to processors based on Intel Core microarchitecture.

**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\_CAPABILITIES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of

IA32\_PERF\_CAPABILITIES are defined in Table 2-2 of Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The relevant bit fields that governs PEBS are:

- 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.
- PEBSaveArchRegs [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.	<ul style="list-style-type: none"> <li>▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set.</li> <li>▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.</li> </ul>	
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: <ul style="list-style-type: none"> <li>▪ Clear all counters if “Counter Freeze on PMI” is not enabled.</li> <li>▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled.</li> </ul> Counters MUST be stopped before writing. <sup>1</sup>	Optional
Disable PEBS.	Clear ENABLE PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write “sample-after” values.	Configure the counter(s) with the sample after value.	

**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.	<ul style="list-style-type: none"> <li>▪ Set local enable bit 22 - 1.</li> <li>▪ Do NOT set local counter PMI/INT bit, bit 20 - 0.</li> <li>▪ Event programmed must be PEBS capable.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Set appropriate OVF_PMI bits - 1.</li> <li>▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.</li> </ul>
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

**NOTES:**

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

**18.6.2.4.4 Re-configuring PEBS Facilities**

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32\_DS\_AREA).

**18.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)**

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32\_MISC\_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32\_DS\_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR\_PEBS\_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 18-70 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_CCCRO	360H	MSR_BSU_ESCR0	7	3A0H
					MSR_FSB_ESCR0	6	3A2H
					MSR_MOB_ESCR0	2	3AAH
					MSR_PMH_ESCR0	4	3ACH
					MSR_BPU_ESCR0	0	3B2H
					MSR_IS_ESCR0	1	3B4H
					MSR_ITLB_ESCR0	3	3B6H
					MSR_IX_ESCR0	5	3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0	7	3A0H
					MSR_FSB_ESCR0	6	3A2H
					MSR_MOB_ESCR0	2	3AAH
					MSR_PMH_ESCR0	4	3ACH
					MSR_BPU_ESCR0	0	3B2H
					MSR_IS_ESCR0	1	3B4H
					MSR_ITLB_ESCR0	3	3B6H
					MSR_IX_ESCR0	5	3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1	7	3A1H
					MSR_FSB_ESCR1	6	3A3H
					MSR_MOB_ESCR1	2	3ABH
					MSR_PMH_ESCR1	4	3ADH
					MSR_BPU_ESCR1	0	3B3H
					MSR_IS_ESCR1	1	3B5H
					MSR_ITLB_ESCR1	3	3B7H
					MSR_IX_ESCR1	5	3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1	7	3A1H
					MSR_FSB_ESCR1	6	3A3H
					MSR_MOB_ESCR1	2	3ABH
					MSR_PMH_ESCR1	4	3ADH
					MSR_BPU_ESCR1	0	3B3H
					MSR_IS_ESCR1	1	3B5H
					MSR_ITLB_ESCR1	3	3B7H
					MSR_IX_ESCR1	5	3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCRO	364H	MSR_MS_ESCR0	0	3C0H
					MSR_TBPU_ESCR0	2	3C2H
					MSR_TC_ESCR0	1	3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0	0	3C0H
					MSR_TBPU_ESCR0	2	3C2H
					MSR_TC_ESCR0	1	3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1	0	3C1H
					MSR_TBPU_ESCR1	2	3C3H
					MSR_TC_ESCR1	1	3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1	0	3C1H
					MSR_TBPU_ESCR1	2	3C3H
					MSR_TC_ESCR1	1	3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCRO	368H	MSR_FIRM_ESCR0	1	3A4H
					MSR_FLAME_ESCR0	0	3A6H
					MSR_DAC_ESCR0	5	3A8H
					MSR_SAAAT_ESCR0	2	3AEH
					MSR_U2L_ESCR0	3	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_SAA_T_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0 <sup>1</sup> 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 <sup>1</sup> 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 <sup>1</sup> 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 <sup>1</sup> 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 <sup>1</sup> 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 <sup>1</sup> 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  $\mu$ ops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting** — A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling** — A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.

- **Processor event-based sampling (PEBS)** — In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

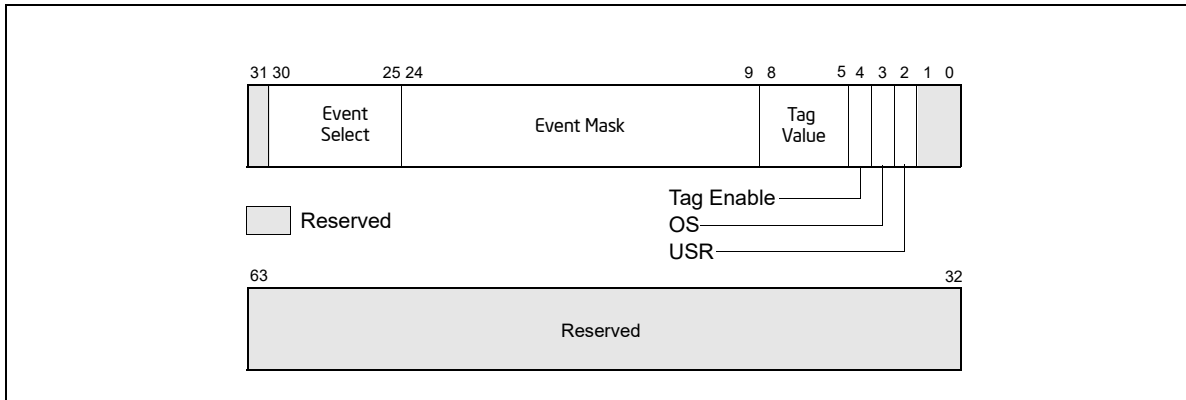
The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

### 18.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-70) 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  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-70 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.

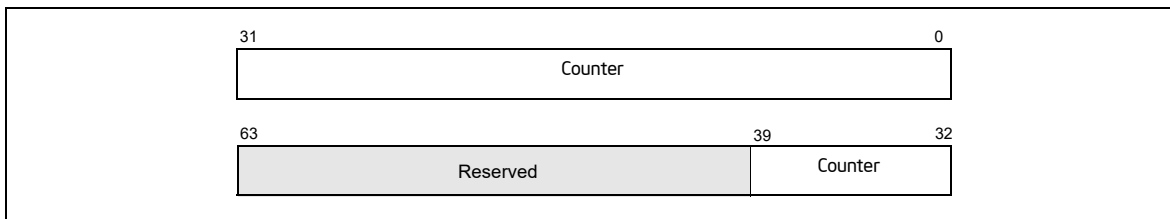
- MSR\_FLAME\_COUNTER2 and MSR\_FLAME\_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR\_IQ\_COUNTER0 and MSR\_IQ\_COUNTER1.
  - MSR\_IQ\_COUNTER2 and MSR\_IQ\_COUNTER3.
  - MSR\_IQ\_COUNTER4 and MSR\_IQ\_COUNTER5.

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.

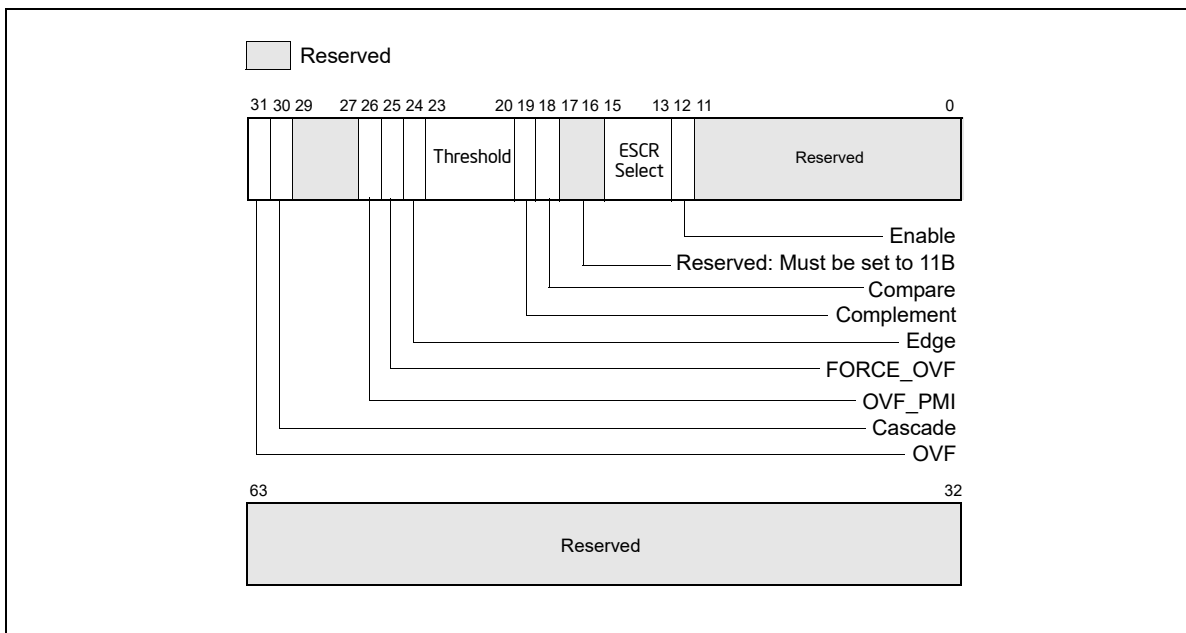
Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement 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 they are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

### 18.6.3.5.1 Selecting Events to Count

Table 19-32 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 2 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counter's threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, then any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

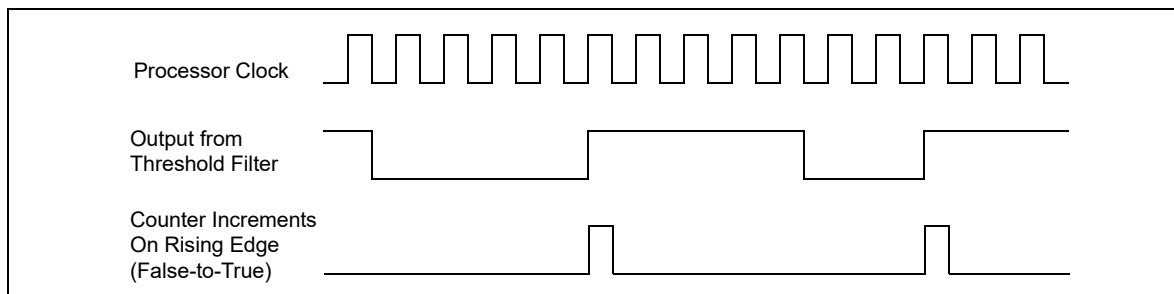
The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 18-48 illustrates rising edge filtering.



The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 18.6.3.5.1, “Selecting Events to Count.”

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
  - Set the compare flag.
  - Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
  - Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 18.6.3.5.3, “Starting Event Counting.”



**Figure 18-48. Effects of Edge Filtering**

### 18.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter’s CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 18.6.3.5.5, “Halting Event Counting”).

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.2, “Filtering Events.”

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 18.6.3.5.4, “Reading a Performance Counter’s Count.”

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 18.6.3.5.6, “Cascading Counters”).

### 18.6.3.5.4 Reading a Performance Counter’s Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 18.6.3.2, “Performance Counters.” These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.3, “Starting Event Counting.”

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-70 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\_MOB\_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	CASCNT4INT00	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INT03	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INT04	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INT05	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_02. For processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_00 and 0F\_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINT0y bits is used.

#### Example 18-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR\_CRU\_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR\_IQ\_CCCR0. This enables CASCNT4INT00 and OVF\_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR\_CRU\_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR\_IQ\_CCCR4 (set ENABLE bit, but not OVF\_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INT0y 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  $\mu$ ops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or  $\mu$ ops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and  $\mu$ ops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction\_Retired and Uops\_Retired in Table 19-32) can count instructions or  $\mu$ ops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging** — Tagging is a means of marking  $\mu$ ops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per  $\mu$ op and a direct count of the event would not provide an indication of how many  $\mu$ ops encountered that event. The tagging mechanisms allow a  $\mu$ op to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per  $\mu$ op, rather than once per

event. For example, a  $\mu$ op may encounter a cache miss more than once during its life time, but a “Miss Retired” metric (that counts the number of retired  $\mu$ ops that encountered a cache miss) will increment only once for that  $\mu$ op. A “Miss Retired” metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, “Related Literature”).

- **Replay** — To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules  $\mu$ ops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied,  $\mu$ ops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of  $\mu$ ops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist** — When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of  $\mu$ ops before they begin and are costly.

### 18.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and  $\mu$ ops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-32, a  $\mu$ op may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count  $\mu$ ops tagged with that mechanism:

- **Front-end tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the `Front_end_event` event.
- **Execution tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered execution events (for example, instruction types) and are counted with the `Execution_Event` event.
- **Replay tagging** — This mechanism pertains to tagging of  $\mu$ ops whose retirement is replayed (for example, a cache miss) and are counted with the `Replay_event` event. Branch mispredictions are also tagged with this mechanism.
- **No tags** — This mechanism does not use tags. It uses the `Instr_retired` and the `Uops_retired` events.

Each tagging mechanism is independent from all others; that is, a  $\mu$ op that has been tagged using one mechanism will not be detected with another mechanism’s tagged- $\mu$ op detector. For example, if  $\mu$ ops are tagged using the front-end tagging mechanisms, the `Replay_event` will not count those as tagged  $\mu$ ops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of  $\mu$ ops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of  $\mu$ ops 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  $\mu$ op and count tagged  $\mu$ ops.

### 18.6.3.6.2 Tagging Mechanism for `Front_end_event`

The `Front_end_event` counts  $\mu$ ops that have been tagged as encountering any of the following events:

- **$\mu$ op decode events** — Tagging  $\mu$ ops for  $\mu$ op decode events requires specifying bits in the `ESCR` associated with the performance-monitoring event, `Uop_type`.
- **Trace cache events** — Tagging  $\mu$ ops 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 MSR's 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  $\mu$ ops that have been tagged with that tag value identifier using `Execution_event` for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular  $\mu$ op. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of `NBOGUS0` should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged  $\mu$ ops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. This mechanism is summarized in the Table 19-35 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  $\mu$ ops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of  $\mu$ op that may experience the replay in the `MSR_PEBS_MATRIX_VERT` MSR and selecting the type of event in the `MSR_PEBS_ENABLE` MSR. Replay tagging must also be enabled with the `UOP_Tag` flag (bit 24) in the `MSR_PEBS_ENABLE` MSR.

The Table 19-36 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 MSR's 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  $\mu$ ops.

### 18.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 17.4.9, "BTS and DS Save Area"). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and `EFLAGS` registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is

generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can only be carried out using the one performance counter, the MSR\_IQ\_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32\_PMC0 and IA32\_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.

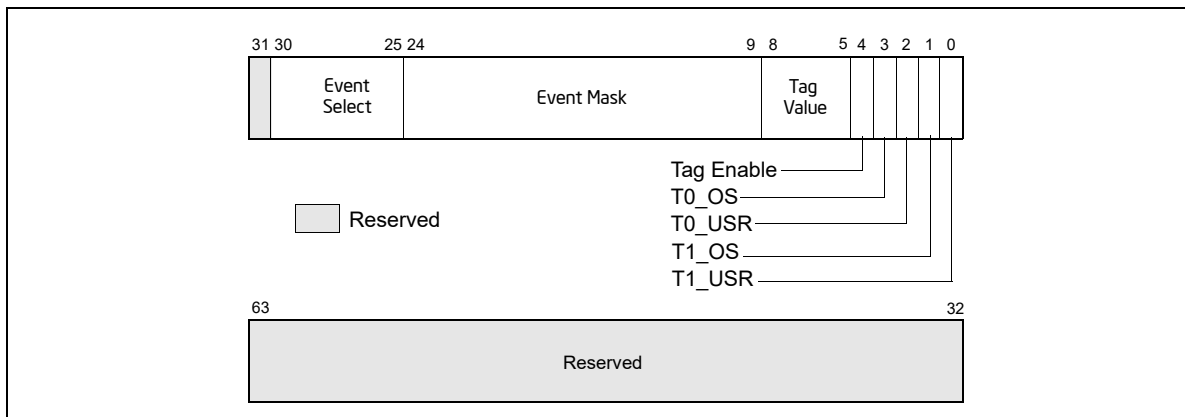


Figure 18-49. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology



- **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  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0\_OS and T0\_USR flags and the T1\_OS and T1\_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, “Identifying Logical Processors in an MP System,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.6.4.4, “Performance Monitoring Events”). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

#### 18.6.4.2 CCCR MSRs

Figure 18-50 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17** — Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:
  - 00 — None. Count only when neither logical processor is active.
  - 01 — Single. Count only when one logical processor is active (either 0 or 1).
  - 10 — Both. Count only when both logical processors are active.
  - 11 — Any. Count when either logical processor is active.
 A halted logical processor or a logical processor in the “wait for SIPI” state is considered inactive.
- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

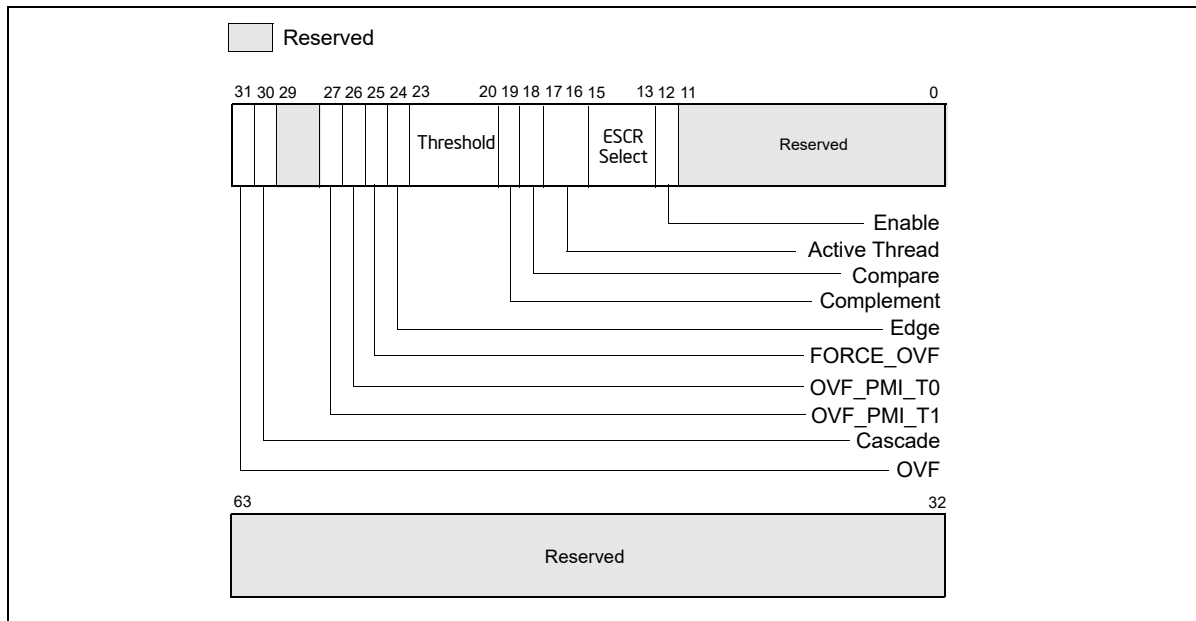


Figure 18-50. Counter Configuration Control Register (CCCR)

- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.6.3.5.2, “Filtering Events”). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.6.3.5.2, “Filtering Events”).
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE\_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF\_PMI\_T0 flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF\_PMI\_T1 flag, bit 27** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.
- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

### 18.6.4.3 IA32\_PEBS\_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR\_PEBS\_ENABLE MSR: bit 25 (ENABLE\_PEBS\_MY\_THR) and 26 (ENABLE\_PEBS\_OTH\_THR) respectively. These bits do not explicitly identify a specific logical processor by logic

processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running (“my thread”) or for the other logical processor in the physical package on which the agent is not running (“other thread”).

PEBS is supported for only a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can be carried out only with two performance counters: MSR\_IQ\_CCCR4 (MSR address 370H) for logical processor 0 and MSR\_IQ\_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE\_PEBS\_MY\_THR and ENABLE\_PEBS\_OTH\_THR bits in the MSR\_PEBS\_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

#### 18.6.4.4 Performance Monitoring Events

All of the events listed in Table 19-31 and 19-32 are available in an Intel Xeon processor MP. When Intel Hyper-Threading Technology is active, many performance monitoring events can be 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) 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

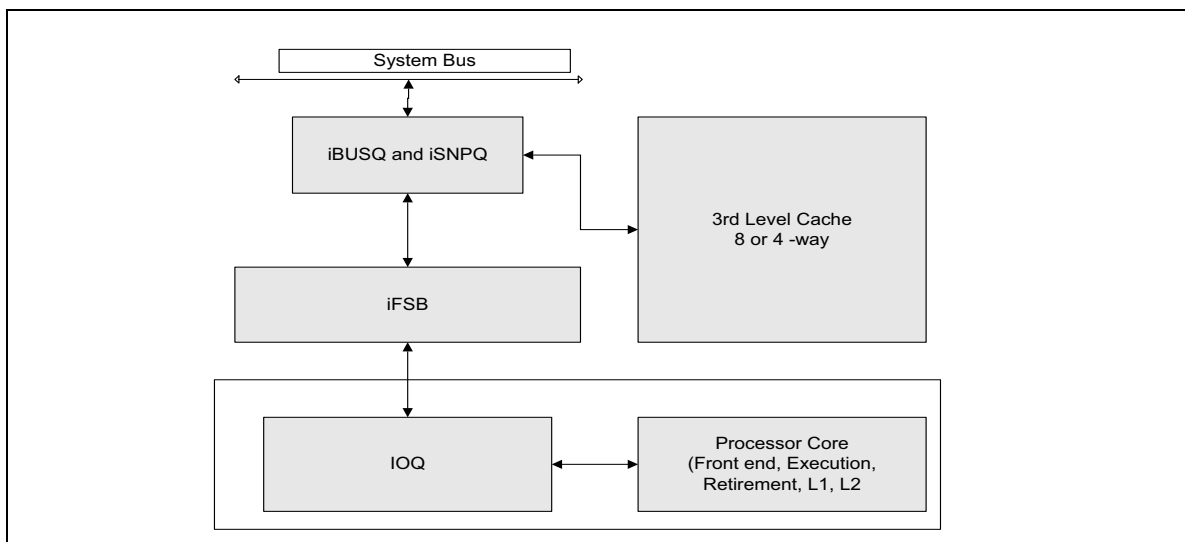
The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

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<sup>®</sup> 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 RDPKC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- IBUSQ event** — This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR\_IFSB\_IBUSQ0 and MSR\_IFSB\_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 18-52.

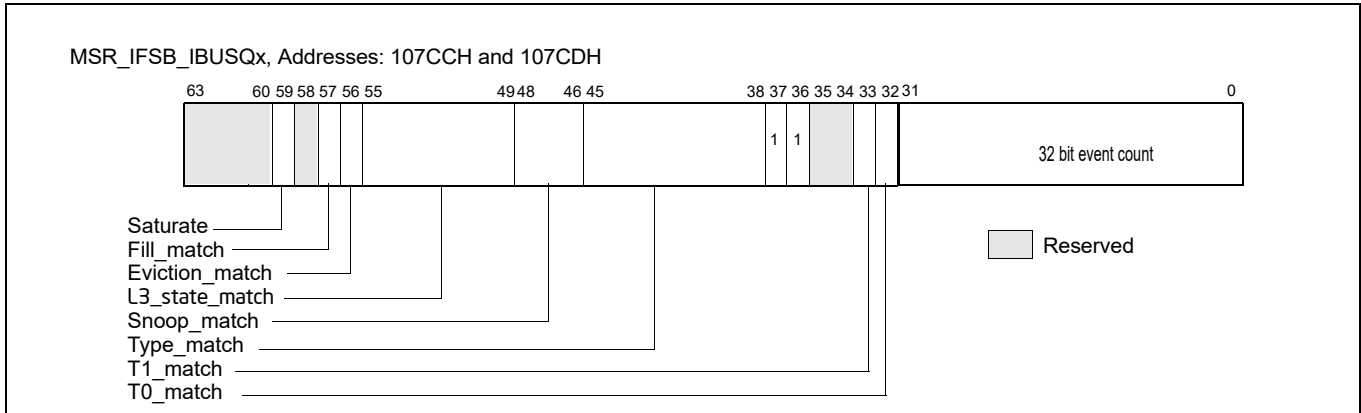


Figure 18-52. MSR\_IFSB\_IBUSQx, Addresses: 107CCH and 107CDH

- ISNPQ event** — This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR\_IFSB\_ISNPQ0 and MSR\_IFSB\_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 18-53.

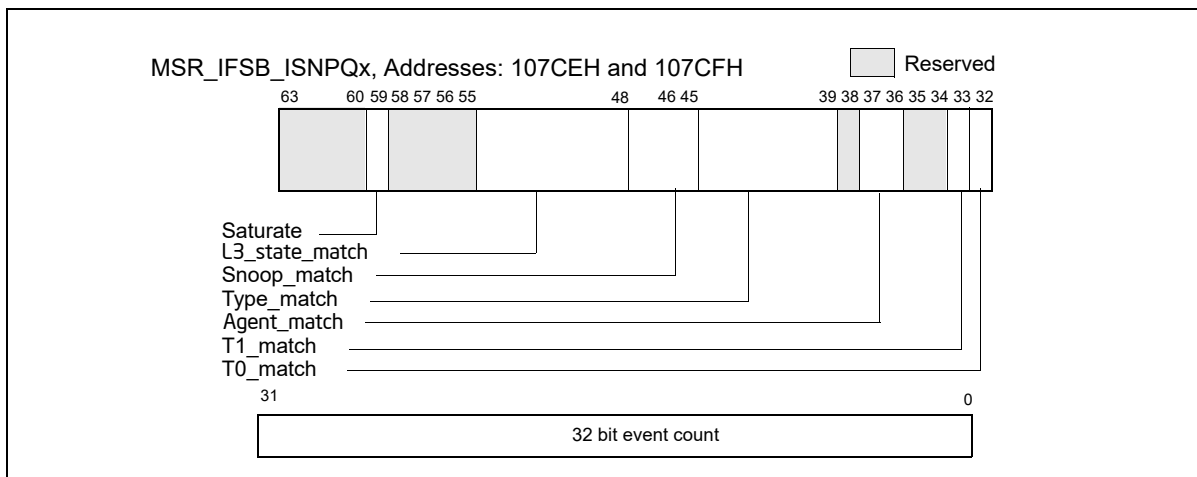


Figure 18-53. MSR\_IFSB\_ISNPQx, Addresses: 107CEH and 107CFH

- EFSB event** — This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR\_EFSB\_DRDY0 and MSR\_EFSB\_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 18-54.

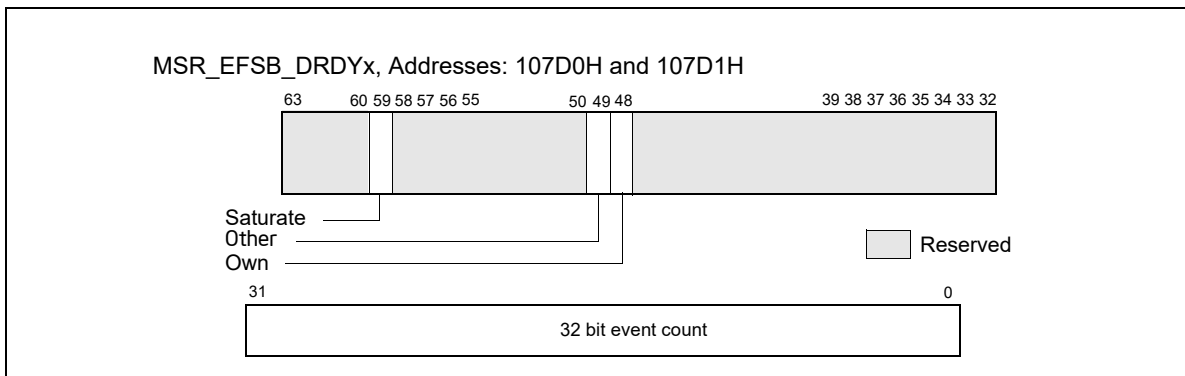


Figure 18-54. MSR\_EFSB\_DRDYx, Addresses: 107D0H and 107D1H

- iBUSQ Latency event** — This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR\_IFSB\_CTRL6[bit 26] to 1; the count freezes after software sets MSR\_IFSB\_CTRL6[bit 26] to 0. MSR\_IFSB\_CNTR7 acts as a 64-bit event counter for this event. See Figure 18-55.

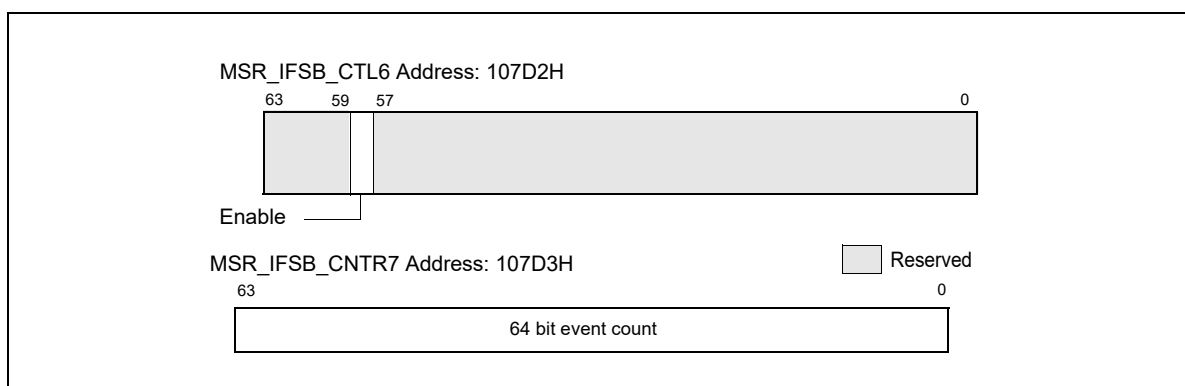


Figure 18-55. MSR\_IFSB\_CTL6, Address: 107D2H;  
 MSR\_IFSB\_CNTR7, Address: 107D3H

## 18.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

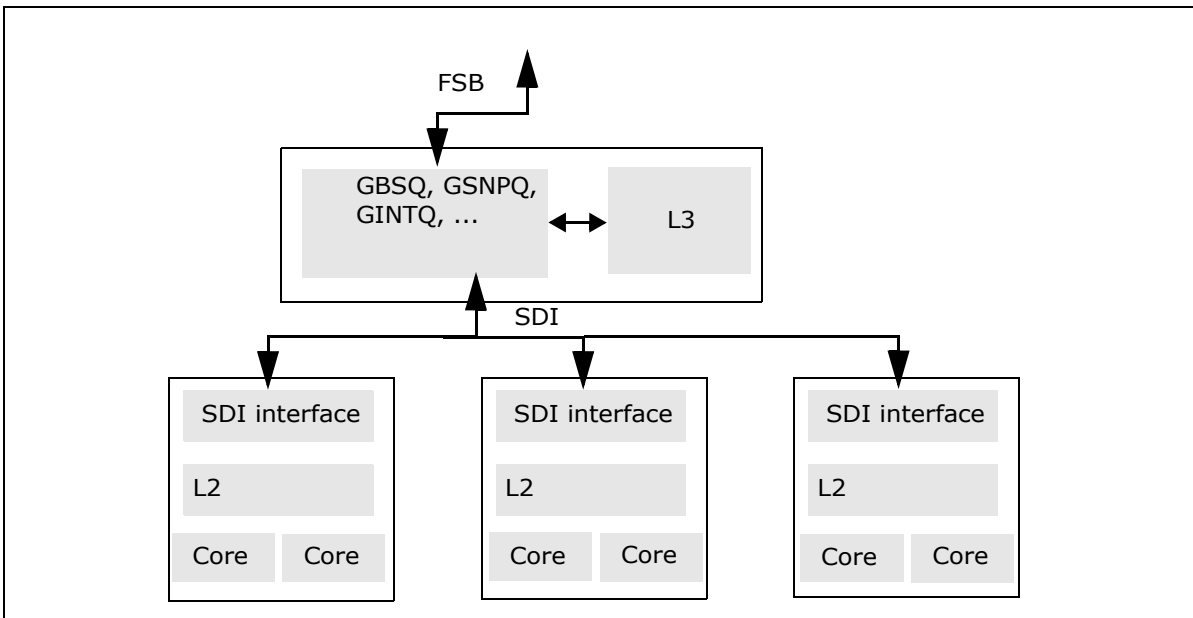
The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily\_DisplayModel value of 06\_1DH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through

additional control logic. See Figure 18-56 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-56 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.



**Figure 18-56. Block Diagram of Intel Xeon Processor 7400 Series**

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.6.4) apply to Intel Xeon processor 7100 series. The MSR's used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily\_DisplayModel signature 06\_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSR's for performance monitoring interface.

The IOQ\_allocation and IOQ\_active\_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.



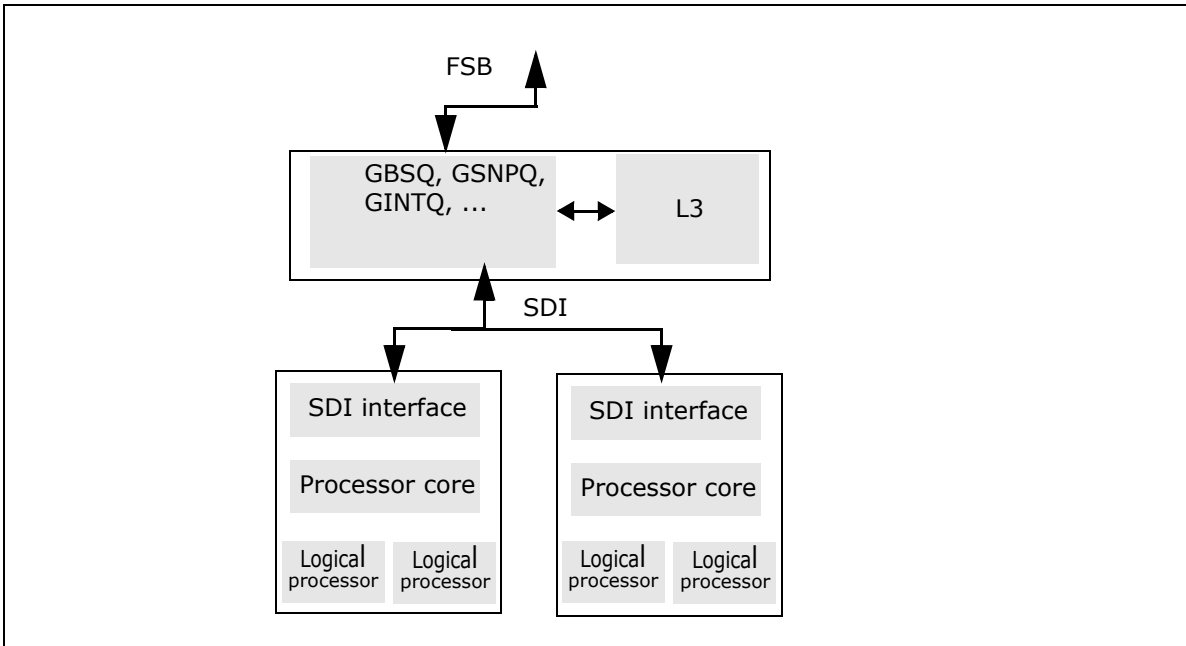


Figure 18-57. Block Diagram of Intel Xeon Processor 7100 Series

### 18.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR\_EMON\_L3\_GL\_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL2 and MSR\_EMON\_L3\_CTR\_CTL3) are dedicated to counting GSNPQ events. Up to two GSNPQ 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

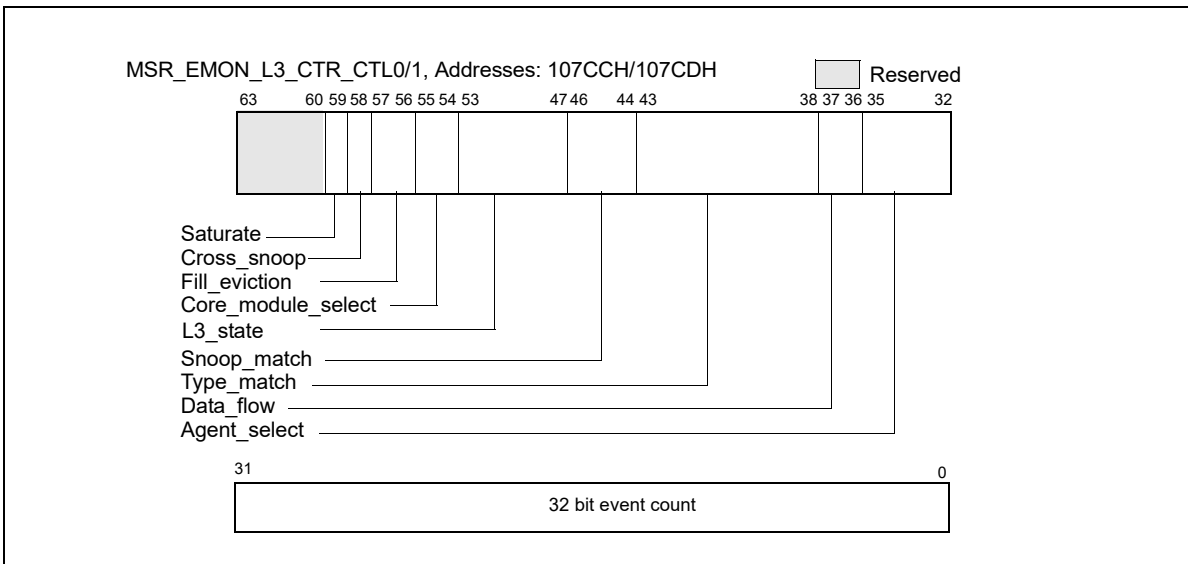
The layout of MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1 is given in Figure 18-58. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

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.

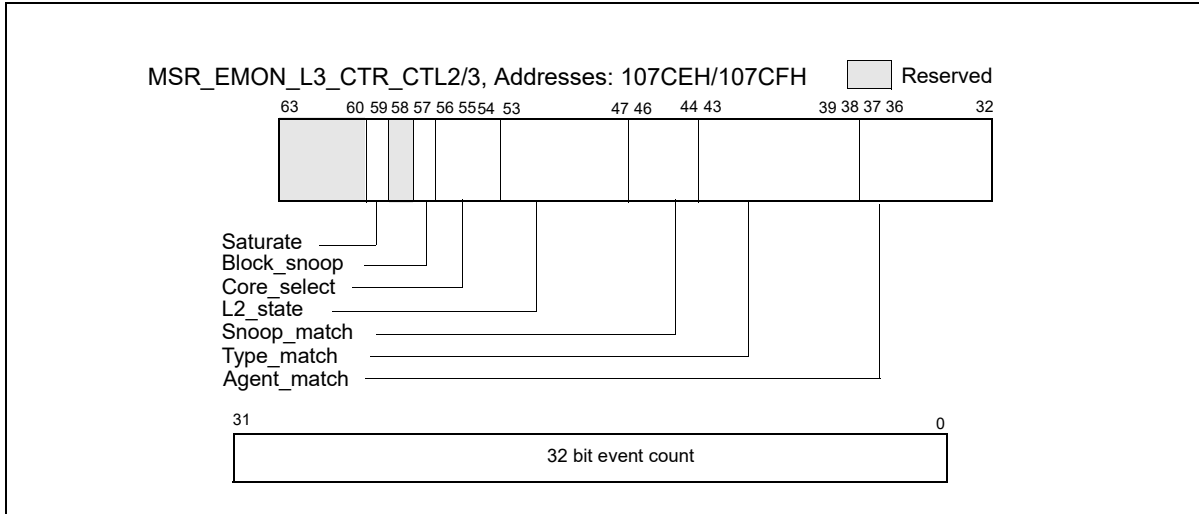


Figure 18-59. MSR\_EMON\_L3\_CTR\_CTL2/3, Addresses: 107CEH/107CFH

### 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.

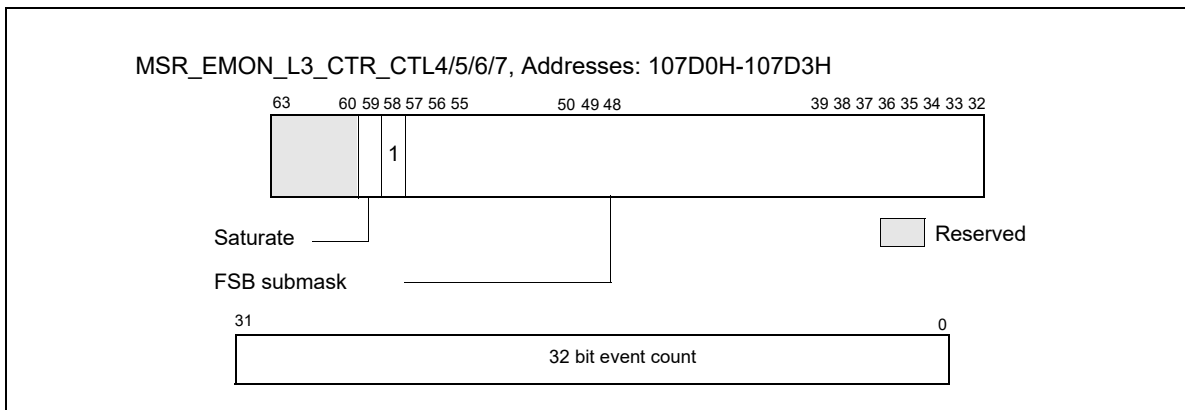


Figure 18-60. MSR\_EMON\_L3\_CTR\_CTL4/5/6/7, Addresses: 107D0H-107D3H

#### 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 MSR: 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**

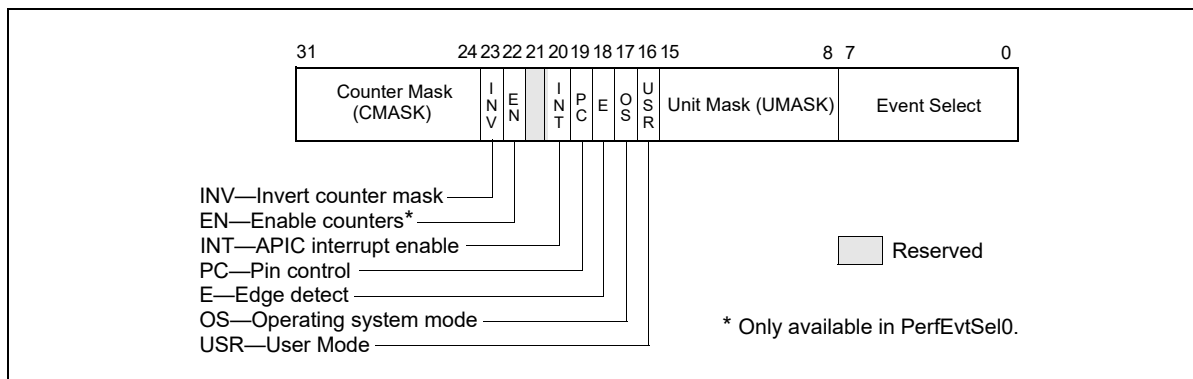
The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed in Table 19-40 are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

**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.

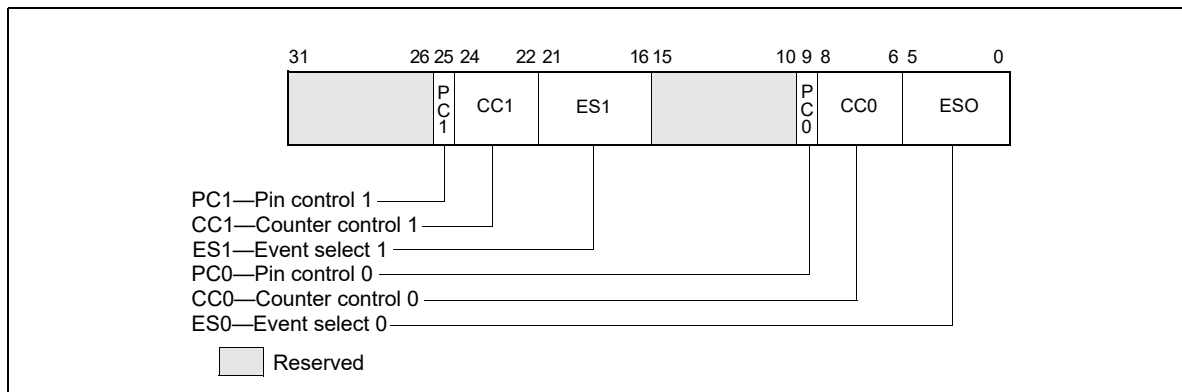


Figure 18-62. CESR MSR (Pentium Processor Only)

- **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 than  $2^{40} - 1$ . After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

### 18.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

## 18.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor’s bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 18.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock** — The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock** — This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock** — These clockticks increment at a fixed frequency and help coordinate the bus on some systems.

- **Core Crystal Clock** — This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks** — Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks** — Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter** — See Section 17.17, “Time-Stamp Counter”.
- **Reference Clockticks** — TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 17.17, “Time-Stamp Counter,” for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI** — Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI** — Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

### 18.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and umask 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

### 18.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 14, “Power and Thermal Management”), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32\_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32\_FIXED\_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32\_MPERF, and IA32\_FIXED\_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

### 18.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 “Information Returned by CPUID Instruction”), the nominal TSC frequency can be determined by using the following equation:

$$\text{Nominal TSC frequency} = (\text{CPUID.15H.ECX}[31:0] * \text{CPUID.15H.EBX}[31:0]) \div \text{CPUID.15H.EAX}[31:0]$$

For Intel processors in which CPUID.15H.EBX[31:0] ÷ 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 <sup>1</sup>	Nominal Core Crystal Clock Frequency
Intel® Xeon® Processor Scalable Family with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.	24 MHz
Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

**NOTES:**

1. For any processor in which CPUID.15H is enumerated and MSR\_PLATFORM\_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

#### 18.7.3.1 For Intel® Processors Based on Microarchitecture Code Name Sandy Bridge, Ivy Bridge, Haswell and Broadwell

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

#### 18.7.3.2 For Intel® Processors Based on Microarchitecture Code Name Nehalem

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

#### 18.7.3.3 For Intel® Atom™ Processors Based on the Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR\_PLATFORM\_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR\_FSB\_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR\_FSB\_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

#### 18.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR\_FSB\_FREQ[2:0] at (0CDH), see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR\_PLATFORM\_ID[12:8]. It corresponds to the Processor Base frequency.
- IF XE operation is enabled, the maximum resolved bus ratio is given in MSR\_PERF\_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR\_PERF\_STATUS[31] is set, XE operation is enabled. The MSR\_PERF\_STATUS[31] field is read-only.

## 18.8 IA32\_PERF\_CAPABILITIES MSR ENUMERATION

The layout of IA32\_PERF\_CAPABILITIES MSR is shown in Figure 18-63, it provides enumeration of a variety of interfaces:

- IA32\_PERF\_CAPABILITIES.LBR\_FMT[bits 5:0]: encodes the LBR format, details are described in Section 17.4.8.1.
- IA32\_PERF\_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist, see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers, see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.PEBS\_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records, see Section 18.6.2.4.2.
- IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[12]: Indicates IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM is supported if 1, see Section 18.8.1.
- IA32\_PERF\_CAPABILITIES.FULL\_WRITE[13]: Indicates the processor supports IA32\_A\_PMCx interface for updating bits 32 and above of IA32\_PMCx, see Section 18.2.5.

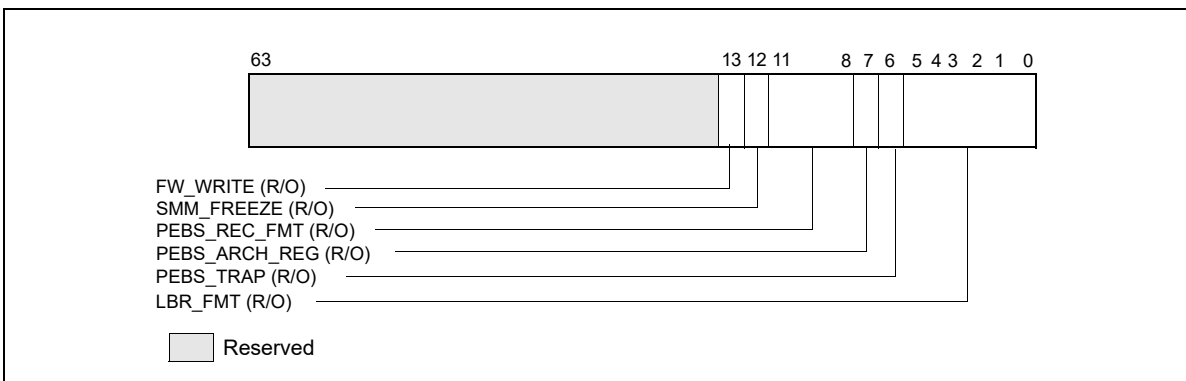


Figure 18-63. Layout of IA32\_PERF\_CAPABILITIES MSR

### 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.

### 13. 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: Updates to Table 24-3 "Format of Interruptibility State". Updates to Section 24.8.3 "VM-Entry Controls for Event Injection" and Section 24.9.2 "Information for VM Exits Due to Vectored Events".

### 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**.<sup>1</sup> 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.<sup>2,3</sup>

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.

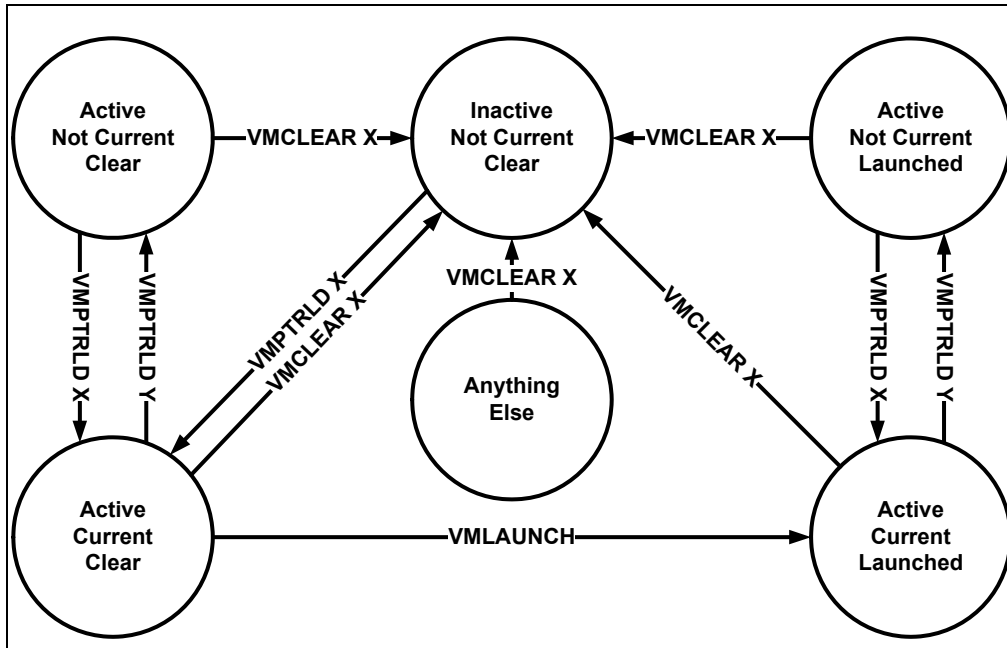


Figure 24-1. States of VMCS X

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.<sup>1</sup> 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.<sup>2</sup> 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.<sup>1</sup> 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<sup>2</sup>. 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.<sup>3</sup>

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).<sup>1</sup>
- 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.<sup>2</sup>
    - 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).<sup>1</sup>

- 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:<sup>2</sup>

- **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**<sup>1</sup> or some other serious error.
- 3: **Wait-for-SIPI**. The logical processor is inactive because it is waiting for a startup-IPI (SIPI).

Future processors may include support for other activity states. Software should read the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine what activity states are supported.

- **Interruptibility state** (32 bits). The IA-32 architecture includes features that permit certain events to be blocked for a period of time. This field contains information about such blocking. Details and the format of this field are given in Table 24-3.

**Table 24-3. Format of Interruptibility State**

Bit Position(s)	Bit Name	Notes
0	Blocking by STI	See the “STI—Set Interrupt Flag” section in Chapter 4 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B</i> . Execution of STI with RFLAGS.IF = 0 blocks maskable interrupts on the instruction boundary following its execution. <sup>1</sup> Setting this bit indicates that this blocking is in effect.
1	Blocking by MOV SS	See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> . Execution of a MOV to SS or a POP to SS blocks or suppresses certain debug exceptions as well as interrupts (maskable and nonmaskable) on the instruction boundary following its execution. Setting this bit indicates that this blocking is in effect. <sup>2</sup> This document uses the term “blocking by MOV SS,” but it applies equally to POP SS.
2	Blocking by SMI	See Section 34.2, “System Management Interrupt (SMI).” System-management interrupts (SMIs) are disabled while the processor is in system-management mode (SMM). Setting this bit indicates that blocking of SMIs is in effect.
3	Blocking by NMI	See Section 6.7.1, “Handling Multiple NMIs,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> and Section 34.8, “NMI Handling While in SMM.” Delivery of a non-maskable interrupt (NMI) or a system-management interrupt (SMI) blocks subsequent NMIs until the next execution of IRET. See Section 25.3 for how this behavior of IRET may change in VMX non-root operation. Setting this bit indicates that blocking of NMIs is in effect. Clearing this bit does not imply that NMIs are not (temporarily) blocked for other reasons. If the “virtual NMIs” VM-execution control (see Section 24.6.1) is 1, this bit does not control the blocking of NMIs. Instead, it refers to “virtual-NMI blocking” (the fact that guest software is not ready for an NMI).
4	Enclave interruption	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.

**NOTES:**

1. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.
2. System-management interrupts may also be inhibited on the instruction boundary following such an execution of MOV or POP.

---

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.

- **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.<sup>1</sup> This field contains information about such exceptions. This field is described in Table 24-4.

**Table 24-4. Format of Pending-Debug-Exceptions**

Bit Position(s)	Bit Name	Notes
3:0	B3 - B0	When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if the corresponding enabling bit in DR7 is not set.
11:4	Reserved	VM entry fails if these bits are not 0. See Section 26.3.1.5.
12	Enabled breakpoint	When set, this bit indicates that at least one data or I/O breakpoint was met and was enabled in DR7.
13	Reserved	VM entry fails if this bit is not 0. See Section 26.3.1.5.
14	BS	When set, this bit indicates that a debug exception would have been triggered by single-step execution mode.
15	Reserved	VM entry fails if this bit is not 0. See Section 26.3.1.5.
16	RTM	When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> ). <sup>1</sup>
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.)

1. For example, execution of a MOV to SS or a POP to SS may inhibit some debug exceptions for one instruction. See Section 6.8.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. In addition, certain events incident to an instruction (for example, an INIT signal) may take priority over debug traps generated by that instruction. See Table 6-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

- **Servicing virtual interrupt (SVI)**. This is the high byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is in service. (The value 0 implies that there is no such interrupt.)

See Chapter 29 for more information on the use of this field.

- **PML index** (16 bits). This field is supported only on processors that support the 1-setting of the “enable PML” VM-execution control. It contains the logical index of the next entry in the page-modification log. Because the page-modification log comprises 512 entries, the PML index is typically a value in the range 0–511. Details of the page-modification log and use of the PML index are given in Section 28.2.5.

## 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 supported only on processors that support the 1-setting of the “load IA32\_PERF\_GLOBAL\_CTRL” VM-exit control.
  - IA32\_PAT (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32\_PAT” VM-exit control.
  - IA32\_EFER (64 bits). This field is supported only on processors that support the 1-setting of the “load IA32\_EFER” VM-exit control.

In addition to the state identified here, some processor state components are loaded with fixed values on every VM exit; there are no fields corresponding to these components in the host-state area. See Section 27.5 for details of how state is loaded on VM exits.

## 24.6 VM-EXECUTION CONTROL FIELDS

The VM-execution control fields govern VMX non-root operation. These are described in Section 24.6.1 through Section 24.6.8.

### 24.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts).<sup>1</sup> Table 24-5 lists the controls. See Chapter 27 for how these controls affect processor behavior in VMX non-root operation.

---

1. Some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls (see Section 25.2).

**Table 24-5. Definitions of Pin-Based VM-Execution Controls**

Bit Position(s)	Name	Description
0	External-interrupt exiting	If this control is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.
3	NMI exiting	If this control is 1, non-maskable interrupts (NMIs) cause VM exits. Otherwise, they are delivered normally using descriptor 2 of the IDT. This control also determines interactions between IRET and blocking by NMI (see Section 25.3).
5	Virtual NMIs	If this control is 1, NMIs are never blocked and the “blocking by NMI” bit (bit 3) in the interruptibility-state field indicates “virtual-NMI blocking” (see Table 24-3). This control also interacts with the “NMI-window exiting” VM-execution control (see Section 24.6.2).
6	Activate VMX-preemption timer	If this control is 1, the VMX-preemption timer counts down in VMX non-root operation; see Section 25.5.1. A VM exit occurs when the timer counts down to zero; see Section 25.2.
7	Process posted interrupts	If this control is 1, the processor treats interrupts with the posted-interrupt notification vector (see Section 24.6.8) specially, updating the virtual-APIC page with posted-interrupt requests (see Section 29.6).

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32\_VMX\_PINBASED\_CTLs and IA32\_VMX\_TRUE\_PINBASED\_CTLs (see Appendix A.3.1) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 26.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 2, and 4. The VMX capability MSR IA32\_VMX\_PINBASED\_CTLs will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32\_VMX\_TRUE\_PINBASED\_CTLs MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

## 24.6.2 Processor-Based VM-Execution Controls

The processor-based VM-execution controls constitute two 32-bit vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions.<sup>1</sup> These are the **primary processor-based VM-execution controls** and the **secondary processor-based VM-execution controls**.

Table 24-6 lists the primary processor-based VM-execution controls. See Chapter 25 for more details of how these controls affect processor behavior in VMX non-root operation.

**Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls**

Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 24.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 24.6.5 and Section 25.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPMC exiting	This control determines whether executions of RDPMC cause VM exits.
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.

1. Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls (see Section 25.1.2), as do task switches (see Section 25.2).



**Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)**

Bit Position(s)	Name	Description
15	CR3-load exiting	In conjunction with the CR3-target controls (see Section 24.6.7), this control determines whether executions of MOV to CR3 cause VM exits. See Section 25.1.3. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
16	CR3-store exiting	This control determines whether executions of MOV from CR3 cause VM exits. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
19	CR8-load exiting	This control determines whether executions of MOV to CR8 cause VM exits.
20	CR8-store exiting	This control determines whether executions of MOV from CR8 cause VM exits.
21	Use TPR shadow	Setting this control to 1 enables TPR virtualization and other APIC-virtualization features. See Chapter 29.
22	NMI-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if there is no virtual-NMI blocking (see Section 24.4.2).
23	MOV-DR exiting	This control determines whether executions of MOV DR cause VM exits.
24	Unconditional I/O exiting	This control determines whether executions of I/O instructions (IN, INS/INSB/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 unpagged protected mode or in real-address mode.
8	APIC-register virtualization	If this control is 1, the logical processor virtualizes certain APIC accesses. See Section 29.4 and Section 29.5.
9	Virtual-interrupt delivery	This control 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\_CTL2 (see Appendix A.3.3) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 26.2.1.1).

### 24.6.3 Exception Bitmap

The **exception bitmap** is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exception's vector.

Whether a page fault (exception with vector 14) causes a VM exit is determined by bit 14 in the exception bitmap as well as the error code produced by the page fault and two 32-bit fields in the VMCS (the **page-fault error-code mask** and **page-fault error-code match**). See Section 25.2 for details.

### 24.6.4 I/O-Bitmap Addresses

The VM-execution control fields include the 64-bit physical addresses of **I/O bitmaps A** and **B** (each of which are 4 KBytes in size). I/O bitmap A contains one bit for each I/O port in the range 0000H through 7FFFH; I/O bitmap B contains bits for ports in the range 8000H through FFFFH.

A logical processor uses these bitmaps if and only if the "use I/O bitmaps" control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if any bit in the I/O bitmaps corresponding to a port it accesses is 1. See Section 25.1.3 for details. If the bitmaps are used, their addresses must be 4-KByte aligned.

### 24.6.5 Time-Stamp Counter Offset and Multiplier

The VM-execution control fields include a 64-bit **TSC-offset** field. If the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1, this field controls executions of the RDTSC and RDTSCP instructions. It also controls executions of the RDMSR instruction that read from the IA32\_TIME\_STAMP\_COUNTER MSR. For all of these, the value of the TSC offset is added to the value of the time-stamp counter, and the sum is returned to guest software in EDX:EAX.

Processors that support the 1-setting of the "use TSC scaling" control also support a 64-bit **TSC-multiplier** field. If this control is 1 (and the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1), this field also affects the executions of the RDTSC, RDTSCP, and RDMSR instructions identified above. Specifically, the contents of the time-stamp counter is first multiplied by the TSC multiplier before adding the TSC offset.

See Chapter 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*).<sup>1</sup>
- If the local APIC is in x2APIC mode, it can access the local APIC's registers using the RDMSR and WRMSR instructions (see *Intel® 64 Architecture Processor Topology Enumeration*).
- In 64-bit mode, it can access the local APIC's task-priority register (TPR) using the MOV CR8 instruction.

There are five processor-based VM-execution controls (see Section 24.6.2) that control such accesses. There are "use TPR shadow", "virtualize APIC accesses", "virtualize x2APIC mode", "virtual-interrupt delivery", and "APIC-register virtualization". These controls interact with the following fields:

- **APIC-access address** (64 bits). This field contains the physical address of the 4-KByte **APIC-access page**. If the "virtualize APIC accesses" VM-execution control is 1, access to this page may cause VM exits or be virtualized by the processor. See Section 29.4.

The APIC-access address exists only on processors that support the 1-setting of the "virtualize APIC accesses" VM-execution control.

- **Virtual-APIC address** (64 bits). This field contains the physical address of the 4-KByte **virtual-APIC page**. The processor uses the virtual-APIC page to virtualize certain accesses to APIC registers and to manage virtual interrupts; see Chapter 29.

Depending on the setting of the controls indicated earlier, the virtual-APIC page may be accessed by the following operations:

- The MOV CR8 instructions (see Section 29.3).
- Accesses to the APIC-access page if, in addition, the "virtualize APIC accesses" VM-execution control is 1 (see Section 29.4).
- The RDMSR and WRMSR instructions if, in addition, the value of ECX is in the range 800H–8FFH (indicating an APIC MSR) and the "virtualize x2APIC mode" VM-execution control is 1 (see Section 29.5).

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 to C0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Write bitmap for low MSRs** (located at the MSR-bitmap address plus 2048). This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.
- **Write bitmap for high MSRs** (located at the MSR-bitmap address plus 3072). This contains one bit for each MSR address in the range C0000000H to C0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

A logical processor uses these bitmaps if and only if the “use MSR bitmaps” control is 1. If the bitmaps are used, an execution of RDMSR or WRMSR causes a VM exit if the value of RCX is in neither of the ranges covered by the bitmaps or if the appropriate bit in the MSR bitmaps (corresponding to the instruction and the RCX value) is 1. See Section 25.1.3 for details. If the bitmaps are used, their address must be 4-KByte aligned.

### 24.6.10 Executive-VMCS Pointer

The executive-VMCS pointer is a 64-bit field used in the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). SMM VM exits save this field as described in Section 34.15.2. VM entries that return from SMM use this field as described in Section 34.15.4.

### 24.6.11 Extended-Page-Table Pointer (EPTP)

The **extended-page-table pointer** (EPTP) contains the address of the base of EPT PML4 table (see Section 28.2.2), as well as other EPT configuration information. The format of this field is shown in Table 24-8.

**Table 24-8. Format of Extended-Page-Table Pointer**

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 28.2.6): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved. <sup>1</sup>
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) <sup>2</sup>
11:7	Reserved
N-1:12	Bits N-1:12 of the physical address of the 4-KByte aligned EPT PML4 table <sup>3</sup>
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. <sup>1</sup> This control must be 0 on processors that do not support Intel 64 architecture.
12	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM exit.
15	Acknowledge interrupt on exit	This control affects VM exits due to external interrupts: <ul style="list-style-type: none"> <li>▪ 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.</li> <li>▪ 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.</li> </ul>
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 CRO.PG and IA32\_EFER.LME, and since CRO.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.<sup>1</sup> 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.<sup>2</sup>
- **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. <sup>1</sup> 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.<sup>1</sup>

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 (e.g., #PF) 4: Software interrupt (INT <i>n</i> ) 5: Privileged software exception (INT1) 6: Software exception (INT3 or INTO) 7: Other event
11	Deliver error code (0 = do not deliver; 1 = deliver)
30:12	Reserved
31	Valid

- The **vector** (bits 7:0) determines which entry in the IDT is used or which other event is injected.
- The **interruption type** (bits 10:8) determines details of how the injection is performed. In general, a VMM should use the type hardware exception for all exceptions **other than** the following:
  - breakpoint exceptions (#BP; a VMM should use the type software exception);
  - overflow exceptions (#OF a VMM should use the use type software exception); and
  - those debug exceptions (#DB) that are generated by INT1 (a VMM should use the use type privileged software exception).<sup>1</sup>

The type **other event** is used for injection of events that are not delivered through the IDT.<sup>2</sup>
- 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.

1. The type hardware exception should be used for all other debug exceptions.

2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with values 1 or 3 for *n*.

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).<sup>1</sup>

### 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; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; and MWAIT. The format of the field depends on the cause of the VM exit. See Section 27.2.1 for details.
- **Guest-linear address** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is used in the following cases:
  - VM exits due to attempts to execute LMSW with a memory operand.
  - VM exits due to attempts to execute INS or OUTS.

1. Software can discover whether these fields can be written by reading the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6).

- VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions.
- Certain VM exits due to EPT violations

See Section 27.2.1 and Section 34.15.2.3 for details of when and how this field is used.

- **Guest-physical address** (64 bits). This field is used VM exits due to EPT violations and EPT misconfigurations. See Section 27.2.1 for details of when and how this field is used.

### 24.9.2 Information for VM Exits Due to Vectored Events

Event-specific information is provided for VM exits due to the following vectored events: exceptions (including those generated by the instructions INT3, INTO, INT1, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). This information is provided in the following fields:

- **VM-exit interruption information** (32 bits). This field receives basic information associated with the event causing the VM exit. Table 24-15 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: Not used 5: Privileged software exception 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
12	NMI unblocking due to IRET
30:13	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.<sup>1</sup> 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 that occur during delivery of hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

See Section 27.2.3 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.<sup>1</sup> 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`.<sup>2</sup> 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).<sup>1</sup> 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 initialize the MSR-bitmap address if the “use MSR bitmaps” VM-execution control is 0.)

A processor maintains some VMCS information that cannot be modified with the VMWRITE instruction; this includes a VMCS’s launch state (see Section 24.1). Such information may be stored in the VMCS data portion of a VMCS region. Because the format of this information is implementation-specific, there is no way for software to know, when it first allocates a region of memory for use as a VMCS region, how the processor will determine this information from the contents of the memory region.

In addition to its other functions, the VMCLEAR instruction initializes any implementation-specific information in the VMCS region referenced by its operand. To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD for the first time. (Figure 24-1 illustrates how execution of VMCLEAR puts a VMCS into a well-defined state.)

The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry for the first time.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since “migrating” a VMCS from one logical processor to another requires use of VMCLEAR (see Section 24.11.1), which sets the launch state of the VMCS to “clear”, such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

### 24.11.4 Software Access to Related Structures

In addition to data in the VMCS region itself, VMX non-root operation can be controlled by data structures that are referenced by pointers in a VMCS (for example, the I/O bitmaps). While the pointers to these data structures are parts of the VMCS, the data structures themselves are not. They are not accessible using VMREAD and VMWRITE but by ordinary memory writes.

Software should ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 24.11.1).

### 24.11.5 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region)<sup>1</sup> 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.<sup>2,3</sup>

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).

## 14. Updates to Chapter 25, Volume 3C

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

-----  
Changes to this chapter: Updates to Section 25.2 "Other Causes of VM Exits", Section 25.4.2 "Treatment of Task Switches", Section 25.5.2 "Monitor Trap Flag", and Section 25.5.5.3 "EPTP Switching".

In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment. This chapter describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation). The differences between VMX non-root operation and ordinary processor operation are described in the following sections:

- Section 25.1, “Instructions That Cause VM Exits”
- Section 25.2, “Other Causes of VM Exits”
- Section 25.3, “Changes to Instruction Behavior in VMX Non-Root Operation”
- Section 25.4, “Other Changes in VMX Non-Root Operation”
- Section 25.5, “Features Specific to VMX Non-Root Operation”
- Section 25.6, “Unrestricted Guests”

Chapter 26, “VM Entries,” describes the data control structures that govern VMX non-root operation. Chapter 26, “VM Entries,” describes the operation of VM entries by which the processor transitions from VMX root operation to VMX non-root operation. Chapter 25, “VMX Non-Root Operation,” describes the operation of VM exits by which the processor transitions from VMX non-root operation to VMX root operation.

Chapter 28, “VMX Support for Address Translation,” describes two features that support address translation in VMX non-root operation. Chapter 29, “APIC Virtualization and Virtual Interrupts,” describes features that support virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC) in VMX non-root operation.

## 25.1 INSTRUCTIONS THAT CAUSE VM EXITS

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are “fault-like,” meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction. Section 27.1 details architectural state in the context of a VM exit.

Section 25.1.1 defines the prioritization between faults and VM exits for instructions subject to both. Section 25.1.2 identifies instructions that cause VM exits whenever they are executed in VMX non-root operation (and thus can never be executed in VMX non-root operation). Section 25.1.3 identifies instructions that cause VM exits depending on the settings of certain VM-execution control fields (see Section 24.6).

### 25.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level,<sup>1</sup> and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.<sup>2</sup>
- Faults incurred while fetching instruction operands have priority over VM exits that are conditioned based on the contents of those operands (see LMSW in Section 25.1.3).
- VM exits caused by execution of the INS and OUTS instructions (resulting either because the “unconditional I/O exiting” VM-execution control is 1 or because the “use I/O bitmaps control is 1”) have priority over the following faults:

---

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.

2. MOV DR is an exception to this rule; see Section 25.1.3.

- A general-protection fault due to the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) being unusable
- A general-protection fault due to an offset beyond the limit of the relevant segment
- An alignment-check exception
- Fault-like VM exits have priority over exceptions other than those mentioned above. For example, RDMSR of a non-existent MSR with CPL = 0 generates a VM exit and not a general-protection exception.

When Section 25.1.2 or Section 25.1.3 (below) identify an instruction execution that may lead to a VM exit, it is assumed that the instruction does not incur a fault that takes priority over a VM exit.

## 25.1.2 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC,<sup>1</sup> INVD, and XSETBV. This is also true of instructions introduced with VMX, which include: INVEPT, INVVPID, VMCALL,<sup>2</sup> VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

## 25.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause “fault-like” VM exits based on the conditions described:<sup>3</sup>

- **CLTS.** The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **ENCLS.** The ENCLS instruction causes a VM exit if the “enable ENCLS exiting” VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLS-exiting bitmap is 1 (see Section 24.6.16).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLS-exiting bitmap is 1.
- **HLT.** The HLT instruction causes a VM exit if the “HLT exiting” VM-execution control is 1.
- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the “unconditional I/O exiting” and “use I/O bitmaps” VM-execution controls:
  - If both controls are 0, the instruction executes normally.
  - If the “unconditional I/O exiting” VM-execution control is 1 and the “use I/O bitmaps” VM-execution control is 0, the instruction causes a VM exit.
  - If the “use I/O bitmaps” VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 24.6.4). If an I/O operation “wraps around” the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the “unconditional I/O exiting” VM-execution control is ignored if the “use I/O bitmaps” VM-execution control is 1).

See Section 25.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.

- **INVLPG.** The INVLPG instruction causes a VM exit if the “INVLPG exiting” VM-execution control is 1.
- **INVPCID.** The INVPCID instruction causes a VM exit if the “INVLPG exiting” and “enable INVPCID” VM-execution controls are both 1.

---

1. An execution of GETSEC in VMX non-root operation causes a VM exit if CR4.SMXE[Bit 14] = 1 regardless of the value of CPL or RAX. An execution of GETSEC causes an invalid-opcode exception (#UD) if CR4.SMXE[Bit 14] = 0.

2. Under the dual-monitor treatment of SMIs and SMM, executions of VMCALL cause SMM VM exits in VMX root operation outside SMM. See Section 34.15.2.

3. Many of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

- **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR.** These instructions cause VM exits if the “descriptor-table exiting” VM-execution control is 1.
- **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  - The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/host mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/host mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
- **MONITOR.** The MONITOR instruction causes a VM exit if the “MONITOR exiting” VM-execution control is 1.
- **MOV from CR3.** The MOV from CR3 instruction causes a VM exit if the “CR3-store exiting” VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
- **MOV from CR8.** The MOV from CR8 instruction causes a VM exit if the “CR8-store exiting” VM-execution control is 1.
- **MOV to CR0.** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
- **MOV to CR3.** The MOV to CR3 instruction causes a VM exit unless the “CR3-load exiting” VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Only the first  $n$  CR3-target values are considered, where  $n$  is the CR3-target count. If the “CR3-load exiting” VM-execution control is 1 and the CR3-target count is 0, MOV to CR3 always causes a VM exit.

The first processors to support the virtual-machine extensions supported only the 1-setting of the “CR3-load exiting” VM-execution control. These processors always consult the CR3-target controls to determine whether an execution of MOV to CR3 causes a VM exit.

- **MOV to CR4.** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- **MOV to CR8.** The MOV to CR8 instruction causes a VM exit if the “CR8-load exiting” VM-execution control is 1.
- **MOV DR.** The MOV DR instruction causes a VM exit if the “MOV-DR exiting” VM-execution control is 1. Such VM exits represent an exception to the principles identified in Section 25.1.1 in that they take priority over the following: general-protection exceptions based on privilege level; and invalid-opcode exceptions that occur because CR4.DE=1 and the instruction specified access to DR4 or DR5.
- **MWAIT.** The MWAIT instruction causes a VM exit if the “MWAIT exiting” VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 25.3).
- **PAUSE.** The behavior of each of this instruction depends on CPL and the settings of the “PAUSE exiting” and “PAUSE-loop exiting” VM-execution controls:

- CPL = 0.

- If the “PAUSE exiting” and “PAUSE-loop exiting” VM-execution controls are both 0, the PAUSE instruction executes normally.
- If the “PAUSE exiting” VM-execution control is 1, the PAUSE instruction causes a VM exit (the “PAUSE-loop exiting” VM-execution control is ignored if CPL = 0 and the “PAUSE exiting” VM-execution control is 1).
- If the “PAUSE exiting” VM-execution control is 0 and the “PAUSE-loop exiting” VM-execution control is 1, the following treatment applies.

The processor determines the amount of time between this execution of PAUSE and the previous execution of PAUSE at CPL 0. If this amount of time exceeds the value of the VM-execution control field PLE\_Gap, the processor considers this execution to be the first execution of PAUSE in a loop. (It also does so for the first execution of PAUSE at CPL 0 after VM entry.)

Otherwise, the processor determines the amount of time since the most recent execution of PAUSE that was considered to be the first in a loop. If this amount of time exceeds the value of the VM-execution control field PLE\_Window, a VM exit occurs.

For purposes of these computations, time is measured based on a counter that runs at the same rate as the timestamp counter (TSC).

- CPL > 0.
  - If the “PAUSE exiting” VM-execution control is 0, the PAUSE instruction executes normally.
  - If the “PAUSE exiting” VM-execution control is 1, the PAUSE instruction causes a VM exit.

The “PAUSE-loop exiting” VM-execution control is ignored if CPL > 0.

- **RDMSR.** The RDMSR instruction causes a VM exit if any of the following are true:
  - The “use MSR bitmaps” VM-execution control is 0.
  - The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in read bitmap for low MSRs is 1, where *n* is the value of ECX.
  - The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in read bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

See Section 24.6.9 for details regarding how these bitmaps are identified.

- **RDPMC.** The RDPMC instruction causes a VM exit if the “RDPMC exiting” VM-execution control is 1.
- **RDRAND.** The RDRAND instruction causes a VM exit if the “RDRAND exiting” VM-execution control is 1.
- **RDSEED.** The RDSEED instruction causes a VM exit if the “RDSEED exiting” VM-execution control is 1.
- **RDTSC.** The RDTSC instruction causes a VM exit if the “RDTSC exiting” VM-execution control is 1.
- **RDTSCP.** The RDTSCP instruction causes a VM exit if the “RDTSC exiting” and “enable RDTSCP” VM-execution controls are both 1.
- **RSM.** The RSM instruction causes a VM exit if executed in system-management mode (SMM).<sup>1</sup>
- **VMREAD.** The VMREAD instruction causes a VM exit if any of the following are true:
  - The “VMCS shadowing” VM-execution control is 0.
  - Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  - Bit *n* in VMREAD bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMREAD bitmap is identified.

If the VMREAD instruction does not cause a VM exit, it reads from the VMCS referenced by the VMCS link pointer. See Chapter 30, “VMREAD—Read Field from Virtual-Machine Control Structure” for details of the operation of the VMREAD instruction.

- **VMWRITE.** The VMWRITE instruction causes a VM exit if any of the following are true:
  - The “VMCS shadowing” VM-execution control is 0.
  - Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  - Bit *n* in VMWRITE bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMWRITE bitmap is identified.

If the VMWRITE instruction does not cause a VM exit, it writes to the VMCS referenced by the VMCS link pointer. See Chapter 30, “VMWRITE—Write Field to Virtual-Machine Control Structure” for details of the operation of the VMWRITE instruction.

- **WBINVD.** The WBINVD instruction causes a VM exit if the “WBINVD exiting” VM-execution control is 1.
- **WRMSR.** The WRMSR instruction causes a VM exit if any of the following are true:
  - The “use MSR bitmaps” VM-execution control is 0.
  - The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in write bitmap for low MSRs is 1, where *n* is the value of ECX.

---

1. Execution of the RSM instruction outside SMM causes an invalid-opcode exception regardless of whether the processor is in VMX operation. It also does so in VMX root operation in SMM; see Section 34.15.3.



- The value of ECX is in the range C0000000H – C0001FFFH and bit  $n$  in write bitmap for high MSRs is 1, where  $n$  is the value of ECX & 00001FFFH.

See Section 24.6.9 for details regarding how these bitmaps are identified.

- **XRSTORS.** The XRSTORS instruction causes a VM exit if the “enable XSAVES/XRSTORS” VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32\_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).
- **XSAVES.** The XSAVES instruction causes a VM exit if the “enable XSAVES/XRSTORS” VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32\_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).

## 25.2 OTHER CAUSES OF VM EXITS

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

- **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 24.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2.<sup>1</sup>

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

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

- **Triple fault.** A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.
- **External interrupts.** An external interrupt causes a VM exit if the “external-interrupt exiting” VM-execution control is 1. (See Section 25.6 for an exception.) Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)
- **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the “NMI exiting” VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
- **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)
- **Start-up IPIs (SIPIs).** SIPIs cause VM exits. If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
- **Task switches.** Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. See Section 25.4.2.

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT  $n$  with value 1 or 3 for  $n$ .

- **System-management interrupts (SMIs)**. If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 34.15.2.<sup>1</sup>
- **VMX-preemption timer**. A VM exit occurs when the timer counts down to zero. See Section 25.5.1 for details of operation of the VMX-preemption timer.

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

These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

- If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS (see Table 24-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.5).

Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.

- If the “NMI-window exiting” VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS (see Table 24-3). (A logical processor may also prevent such a VM exit if there is blocking of events by STI.) Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.6).

VM exits caused by the VMX-preemption timer and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

## 25.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:<sup>2</sup>

- **CLTS**. Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 23.8), in which case CLTS causes a general-protection exception.
  - If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.
- **INVPCID**. Behavior of the INVPCID instruction is determined first by the setting of the “enable INVPCID” VM-execution control:

1. Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 34.14.1.

2. Some of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

- If the “enable INVPCID” VM-execution control is 0, INVPCID causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
- If the “enable INVPCID” VM-execution control is 1, treatment is based on the setting of the “INVLPG exiting” VM-execution control:
  - If the “INVLPG exiting” VM-execution control is 0, INVPCID operates normally.
  - If the “INVLPG exiting” VM-execution control is 1, INVPCID causes a VM exit.
- **IRET.** Behavior of IRET with regard to NMI blocking (see Table 24-3) is determined by the settings of the “NMI exiting” and “virtual NMIs” VM-execution controls:
  - If the “NMI exiting” VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” control must be 0; see Section 26.2.1.1.)
  - If the “NMI exiting” VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the “virtual NMIs” VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.

- **LMSW.** Outside of VMX non-root operation, LMSW loads its source operand into CR0[3:0], but it does not clear CR0.PE if that bit is set. In VMX non-root operation, an execution of LMSW that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0[3:0] corresponding to a bit set in the CR0 guest/host mask. An attempt to set any other bit in CR0[3:0] to a value not supported in VMX operation (see Section 23.8) causes a general-protection exception. Attempts to clear CR0.PE are ignored without fault.
- **MOV from CR0.** The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.  
Depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **MOV from CR3.** If the “enable EPT” VM-execution control is 1 and an execution of MOV from CR3 does not cause a VM exit (see Section 25.1.3), the value loaded from CR3 is a guest-physical address; see Section 28.2.1.
- **MOV from CR4.** The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow.  
Depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.
- **MOV from CR8.** If the MOV from CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the “use TPR shadow” VM-execution control is 1; see Section 29.3.
- **MOV to CR0.** An execution of MOV to CR0 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. Treatment of attempts to modify other bits in CR0 depends on the setting of the “unrestricted guest” VM-execution control:
  - If the control is 0, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 to a value not supported in VMX operation (see Section 23.8).
  - If the control is 1, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 other than bit 0 (PE) or bit 31 (PG) to a value not supported in VMX operation. It remains the case, however, that MOV to CR0 causes a general-protection exception if it would result in CR0.PE = 0 and CR0.PG = 1 or if it would result in CR0.PG = 1, CR4.PAE = 0, and IA32\_EFER.LME = 1.
- **MOV to CR3.** If the “enable EPT” VM-execution control is 1 and an execution of MOV to CR3 does not cause a VM exit (see Section 25.1.3), the value loaded into CR3 is treated as a guest-physical address; see Section 28.2.1.

- If PAE paging is not being used, the instruction does not use the guest-physical address to access memory and it does not cause it to be translated through EPT.<sup>1</sup>
- If PAE paging is being used, the instruction translates the guest-physical address through EPT and uses the result to load the four (4) page-directory-pointer-table entries (PDPTEs). The instruction does not use the guest-physical addresses the PDPTEs to access memory and it does not cause them to be translated through EPT.
- **MOV to CR4.** An execution of MOV to CR4 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit in CR4 (not corresponding to a bit set in the CR4 guest/host mask) to a value not supported in VMX operation (see Section 23.8).
- **MOV to CR8.** If the MOV to CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the “use TPR shadow” VM-execution control is 1; see Section 29.3.
- **MWAIT.** Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the “MWAIT exiting” VM-execution control:
  - If the “MWAIT exiting” VM-execution control is 1, MWAIT causes a VM exit.
  - If the “MWAIT exiting” VM-execution control is 0, MWAIT operates normally if one of the following are true: (1) ECX[0] is 0; (2) RFLAGS.IF = 1; or both of the following are true: (a) the “interrupt-window exiting” VM-execution control is 0; and (b) the logical processor has not recognized a pending virtual interrupt (see Section 29.2.1).
  - If the “MWAIT exiting” VM-execution control is 0, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state if either the “interrupt-window exiting” VM-execution control is 1 or the logical processor has recognized a pending virtual interrupt; instead, control passes to the instruction following the MWAIT instruction.
- **RDMSR.** Section 25.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction’s behavior may be modified for certain values of ECX:
  - If ECX contains 10H (indicating the IA32\_TIME\_STAMP\_COUNTER MSR), the value returned by the instruction is determined by the setting of the “use TSC offsetting” VM-execution control:
    - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32\_TIME\_STAMP\_COUNTER MSR.
    - If the control is 1, the value returned is determined by the setting of the “use TSC scaling” VM-execution control:
      - If the control is 0, RDMSR loads EAX:EDX with the sum of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC offset.
      - If the control is 1, RDMSR first computes the product of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

The 1-setting of the “use TSC-offsetting” VM-execution control does not affect executions of RDMSR if ECX contains 6E0H (indicating the IA32\_TSC\_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.

  - If ECX is in the range 800H–8FFH (indicating an APIC MSR), instruction behavior may be modified if the “virtualize x2APIC mode” VM-execution control is 1; see Section 29.5.- **RDPID.** Behavior of the RDPID instruction is determined first by the setting of the “enable RDTSCP” VM-execution control:
  - If the “enable RDTSCP” VM-execution control is 0, RDPID causes an invalid-opcode exception (#UD).
  - If the “enable RDTSCP” VM-execution control is 1, RDPID operates normally.

---

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32\_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

- **RDTSC.** Behavior of the RDTSC instruction is determined by the settings of the “RDTSC exiting” and “use TSC offsetting” VM-execution controls:
  - If both controls are 0, RDTSC operates normally.
  - If the “RDTSC exiting” VM-execution control is 0 and the “use TSC offsetting” VM-execution control is 1, the value returned is determined by the setting of the “use TSC scaling” VM-execution control:
    - If the control is 0, RDTSC loads EAX:EDX with the sum of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC offset.
    - If the control is 1, RDTSC first computes the product of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.
  - If the “RDTSC exiting” VM-execution control is 1, RDTSC causes a VM exit.
- **RDTSCP.** Behavior of the RDTSCP instruction is determined first by the setting of the “enable RDTSCP” VM-execution control:
  - If the “enable RDTSCP” VM-execution control is 0, RDTSCP causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
  - If the “enable RDTSCP” VM-execution control is 1, treatment is based on the settings of the “RDTSC exiting” and “use TSC offsetting” VM-execution controls:
    - If both controls are 0, RDTSCP operates normally.
    - If the “RDTSC exiting” VM-execution control is 0 and the “use TSC offsetting” VM-execution control is 1, the value returned is determined by the setting of the “use TSC scaling” VM-execution control:
      - If the control is 0, RDTSCP loads EAX:EDX with the sum of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC offset.
      - If the control is 1, RDTSCP first computes the product of the value of the IA32\_TIME\_STAMP\_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

In either case, RDTSCP also loads ECX with the value of bits 31:0 of the IA32\_TSC\_AUX MSR.

  - If the “RDTSC exiting” VM-execution control is 1, RDTSCP causes a VM exit.
- **SMSW.** The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, SMSW reads normally from CR0; if every bit is set in the CR0 guest/host mask, SMSW returns the value of the CR0 read shadow.
 

Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **WRMSR.** Section 25.1.3 identifies when executions of the WRMSR instruction cause VM exits. If such an execution neither a fault due to CPL > 0 nor a VM exit, the instruction’s behavior may be modified for certain values of ECX:
  - If ECX contains 79H (indicating IA32\_BIOS\_UPDT\_TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.
  - On processors that support Intel PT but which do not allow it to be used in VMX operation, if ECX contains 570H (indicating the IA32\_RTIT\_CTL MSR), the instruction causes a general-protection exception.<sup>1</sup>
  - If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), or 83FH (self-IPI MSR), instruction behavior may be modified if the “virtualize x2APIC mode” VM-execution control is 1; see Section 29.5.

- **XRSTORS.** Behavior of the XRSTORS instruction is determined first by the setting of the “enable XSAVES/XRSTORS” VM-execution control:
  - If the “enable XSAVES/XRSTORS” VM-execution control is 0, XRSTORS causes an invalid-opcode exception (#UD).
  - If the “enable XSAVES/XRSTORS” VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XRSTORS causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32\_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XRSTORS operates normally.
- **XSAVES.** Behavior of the XSAVES instruction is determined first by the setting of the “enable XSAVES/XRSTORS” VM-execution control:
  - If the “enable XSAVES/XRSTORS” VM-execution control is 0, XSAVES causes an invalid-opcode exception (#UD).
  - If the “enable XSAVES/XRSTORS” VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XSAVES causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32\_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XSAVES operates normally.

## 25.4 OTHER CHANGES IN VMX NON-ROOT OPERATION

Treatments of event blocking and of task switches differ in VMX non-root operation as described in the following sections.

### 25.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the “external-interrupt exiting” VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0).
- If the “external-interrupt exiting” VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).
- If the “NMI exiting” VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).

### 25.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:
  - a. If CALL, INT *n*, INT1, INT3, INTO, or JMP accesses a task gate in IA-32e mode, a general-protection exception occurs.

---

1. Software should read the VMX capability MSR IA32\_VMX\_MISC to determine whether the processor allows Intel PT to be used in VMX operation (see Appendix A.6).

- b. If CALL, INT  $n$ , INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.
  - c. If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general-protection exception occurs.
  - d. If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.
  - e. If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.
  - f. If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.
  - g. If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs.
  - h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.
2. Checks are made on the new TSS selector (for example, that is within GDT limits).
  3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).
  4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.

Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

If an attempt at a task switch through a task gate in the IDT causes an exception (before generating a VM exit due to the task switch) and that exception causes a VM exit, information about the event whose delivery that accessed the task gate is recorded in the IDT-vectoring information fields and information about the exception that caused the VM exit is recorded in the VM-exit interruption-information fields. See Section 27.2. The fact that a task gate was being accessed is not recorded in the VMCS.

If an attempt at a task switch through a task gate in the IDT causes VM exit due to the task switch, information about the event whose delivery accessed the task gate is recorded in the IDT-vectoring fields of the VMCS. Since the cause of such a VM exit is a task switch and not an interruption, the valid bit for the VM-exit interruption information field is 0. See Section 27.2.

## 25.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION

Some VM-execution controls support features that are specific to VMX non-root operation. These are the VMX-preemption timer (Section 25.5.1) and the monitor trap flag (Section 25.5.2), translation of guest-physical addresses (Section 25.5.3), VM functions (Section 25.5.5), and virtualization exceptions (Section 25.5.6).

### 25.5.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of “activate VMX-preemption timer” VM-execution control, the **VMX-preemption timer** counts down (from the value loaded by VM entry; see Section 26.6.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 25.2).

The VMX-preemption timer counts down at rate proportional to that of the timestamp counter (TSC). Specifically, the timer counts down by 1 every time bit X in the TSC changes due to a TSC increment. The value of X is in the range 0–31 and can be determined by consulting the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6).

The VMX-preemption timer operates in the C-states C0, C1, and C2; it also operates in the shutdown and wait-for-SIPI states. If the timer counts down to zero in any state other than the wait-for SIPI state, the logical processor

transitions to the C0 C-state and causes a VM exit; the timer does not cause a VM exit if it counts down to zero in the wait-for-SIPI state. The timer is not decremented in C-states deeper than C2.

Treatment of the timer in the case of system management interrupts (SMIs) and system-management mode (SMM) depends on whether the treatment of SMIs and SMM:

- If the default treatment of SMIs and SMM (see Section 34.14) is active, the VMX-preemption timer counts across an SMI to VMX non-root operation, subsequent execution in SMM, and the return from SMM via the RSM instruction. However, the timer can cause a VM exit only from VMX non-root operation. If the timer expires during SMI, in SMM, or during RSM, a timer-induced VM exit occurs immediately after RSM with its normal priority unless it is blocked based on activity state (Section 25.2).
- If the dual-monitor treatment of SMIs and SMM (see Section 34.15) is active, transitions into and out of SMM are VM exits and VM entries, respectively. The treatment of the VMX-preemption timer by those transitions is mostly the same as for ordinary VM exits and VM entries; Section 34.15.2 and Section 34.15.4 detail some differences.

## 25.5.2 Monitor Trap Flag

The **monitor trap flag** is a debugging feature that causes VM exits to occur on certain instruction boundaries in VMX non-root operation. Such VM exits are called **MTF VM exits**. An MTF VM exit may occur on an instruction boundary in VMX non-root operation as follows:

- If the “monitor trap flag” VM-execution control is 1 and VM entry is injecting a vectored event (see Section 26.5.1), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry.
- If VM entry is injecting a pending MTF VM exit (see Section 26.5.2), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry. This is the case even if the “monitor trap flag” VM-execution control is 0.
- If the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and a pending event (e.g., debug exception or interrupt) is delivered before an instruction can execute, an MTF VM exit is pending on the instruction boundary following delivery of the event (or any nested exception).
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is a REP-prefixed string instruction:
  - If the first iteration of the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the first iteration of the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary after that iteration.
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is the XBEGIN instruction. In this case, an MTF VM exit is pending at the fallback instruction address of the XBEGIN instruction. This behavior applies regardless of whether advanced debugging of RTM transactional regions has been enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is neither a REP-prefixed string instruction or the XBEGIN instruction:
  - If the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).<sup>1</sup>
  - If the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary following execution of that instruction. If the instruction is INT1, INT3, or INTO, this boundary follows delivery of any software exception. If the instruction is INT *n*, this boundary follows delivery of a software interrupt. If the instruction is HLT, the MTF VM exit will be from the HLT activity state.

No MTF VM exit occurs if another VM exit occurs before reaching the instruction boundary on which an MTF VM exit would be pending (e.g., due to an exception or triple fault).

1. This item includes the cases of an invalid opcode exception—#UD— generated by the UDO, UD1, and UD2 instructions and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.



An MTF VM exit occurs on the instruction boundary on which it is pending unless a higher priority event takes precedence or the MTF VM exit is blocked due to the activity state:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over MTF VM exits. MTF VM exits take priority over debug-trap exceptions and lower priority events.
- No MTF VM exit occurs if the processor is in either the shutdown activity state or wait-for-SIPI activity state. If a non-maskable interrupt subsequently takes the logical processor out of the shutdown activity state without causing a VM exit, an MTF VM exit is pending after delivery of that interrupt.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 42.2 for details.

### 25.5.3 Translation of Guest-Physical Addresses Using EPT

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain physical addresses are treated as guest-physical addresses and are not used to access memory directly. Instead, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

Details of the EPT mechanism are given in Section 28.2.

### 25.5.4 APIC Virtualization

APIC virtualization is a collection of features that can be used to support the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC). When APIC virtualization is enabled, the processor emulates many accesses to the APIC, tracks the state of the virtual APIC, and delivers virtual interrupts — all in VMX non-root operation without a VM exit.

Details of the APIC virtualization are given in Chapter 29.

### 25.5.5 VM Functions

A **VM function** is an operation provided by the processor that can be invoked from VMX non-root operation without a VM exit. VM functions are enabled and configured by the settings of different fields in the VMCS. Software in VMX non-root operation invokes a VM function with the VMFUNC instruction; the value of EAX selects the specific VM function being invoked.

Section 25.5.5.1 explains how VM functions are enabled. Section 25.5.5.2 specifies the behavior of the VMFUNC instruction. Section 25.5.5.3 describes a specific VM function called EPTP switching.

#### 25.5.5.1 Enabling VM Functions

Software enables VM functions generally by setting the “enable VM functions” VM-execution control. A specific VM function is enabled by setting the corresponding VM-function control.

Suppose, for example, that software wants to enable EPTP switching (VM function 0; see Section 24.6.14). To do so, it must set the “activate secondary controls” VM-execution control (bit 31 of the primary processor-based VM-execution controls), the “enable VM functions” VM-execution control (bit 13 of the secondary processor-based VM-execution controls) and the “EPTP switching” VM-function control (bit 0 of the VM-function controls).

#### 25.5.5.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (#UD) if the “enable VM functions” VM-execution controls is 0<sup>1</sup> or the value of EAX is greater than 63 (only VM functions 0–63 can be enable). Otherwise, the instruction causes a VM exit if the bit at position EAX is 0 in the VM-function controls (the selected VM function is

1. “Enable VM functions” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “enable VM functions” VM-execution control were 0. See Section 24.6.2.

not enabled). If such a VM exit occurs, the basic exit reason used is 59 (3BH), indicating “VMFUNC”, and the length of the VMFUNC instruction is saved into the VM-exit instruction-length field. If the instruction causes neither an invalid-opcode exception nor a VM exit due to a disabled VM function, it performs the functionality of the VM function specified by the value in EAX.

Individual VM functions may perform additional fault checking (e.g., one might cause a general-protection exception if  $CPL > 0$ ). In addition, specific VM functions may include checks that might result in a VM exit. If such a VM exit occurs, VM-exit information is saved as described in the previous paragraph. The specification of a VM function may indicate that additional VM-exit information is provided.

The specific behavior of the EPTP-switching VM function (including checks that result in VM exits) is given in Section 25.5.5.3.

### 25.5.5.3 EPTP Switching

EPTP switching is VM function 0. This VM function allows software in VMX non-root operation to load a new value for the EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy (see Section 28.2 for details of the operation of EPT). Software is limited to selecting from a list of potential EPTP values configured in advance by software in VMX root operation.

Specifically, the value of ECX is used to select an entry from the EPTP list, the 4-KByte structure referenced by the EPTP-list address (see Section 24.6.14; because this structure contains 512 8-Byte entries, VMFUNC causes a VM exit if  $ECX \geq 512$ ). If the selected entry is a valid EPTP value (it would not cause VM entry to fail; see Section 26.2.1.1), it is stored in the EPTP field of the current VMCS and is used for subsequent accesses using guest-physical addresses. The following pseudocode provides details:

```

IF ECX ≥ 512
    THEN VM exit;
ELSE
    tent_EPTP ← 8 bytes from EPTP-list address + 8 * ECX;
    IF tent_EPTP is not a valid EPTP value (would cause VM entry to fail if in EPTP)
        THEN VM exit;
    ELSE
        write tent_EPTP to the EPTP field in the current VMCS;
        use tent_EPTP as the new EPTP value for address translation;
        IF processor supports the 1-setting of the “EPT-violation #VE” VM-execution control
            THEN
                write ECX[15:0] to EPTP-index field in current VMCS;
                use ECX[15:0] as EPTP index for subsequent EPT-violation virtualization exceptions (see Section 25.5.6.2);
        FI;
    FI;
FI;

```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

As noted in Section 25.5.5.2, an execution of the EPTP-switching VM function that causes a VM exit (as specified above), uses the basic exit reason 59, indicating “VMFUNC”. The length of the VMFUNC instruction is saved into the VM-exit instruction-length field. No additional VM-exit information is provided.

An execution of VMFUNC loads EPTP from the EPTP list (and thus does not cause a fault or VM exit) is called an **EPTP-switching VMFUNC**. After an EPTP-switching VMFUNC, control passes to the next instruction. The logical processor starts creating and using guest-physical and combined mappings associated with the new value of bits 51:12 of EPTP; the combined mappings created and used are associated with the current VPID and PCID (these are not changed by VMFUNC).<sup>1</sup> If the “enable VPID” VM-execution control is 0, an EPTP-switching VMFUNC invalidates combined mappings associated with VPID 0000H (for all PCIDs and for all EP4TA values, where EP4TA is the value of bits 51:12 of EPTP).

Because an EPTP-switching VMFUNC may change the translation of guest-physical addresses, it may affect use of the guest-physical address in CR3. The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT viola-

1. If the “enable VPID” VM-execution control is 0, the current VPID is 0000H; if  $CR4.PCIDE = 0$ , the current PCID is 000H.

tion or an EPT misconfiguration due to the translation of that guest-physical address through the new EPT paging structures. The following items provide details that apply if CR0.PG = 1:

- If 32-bit paging or 4-level paging<sup>1</sup> is in use (either CR4.PAE = 0 or IA32\_EFER.LMA = 1), the next memory access with a linear address uses the translation of the guest-physical address in CR3 through the new EPT paging structures. As a result, this access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.
- If PAE paging is in use (CR4.PAE = 1 and IA32\_EFER.LMA = 0), an EPT-switching VMFUNC does not load the four page-directory-pointer-table entries (PDPTes) from the guest-physical address in CR3. The logical processor continues to use the four guest-physical addresses already present in the PDPTes. The guest-physical address in CR3 is not translated through the new EPT paging structures (until some operation that would load the PDPTes).

The EPT-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during the translation of a guest-physical address in any of the PDPTes. A subsequent memory access with a linear address uses the translation of the guest-physical address in the appropriate PDPTE through the new EPT paging structures. As a result, such an access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.

If an EPT-switching VMFUNC establishes an EPTP value that enables accessed and dirty flags for EPT (by setting bit 6), subsequent memory accesses may fail to set those flags as specified if there has been no appropriate execution of INVEPT since the last use of an EPTP value that does not enable accessed and dirty flags for EPT (because bit 6 is clear) and that is identical to the new value on bits 51:12.

If the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control, an EPT-switching VMFUNC loads the value in ECX[15:0] into to EPTP-index field in current VMCS. Subsequent EPT-violation virtualization exceptions will save this value into the virtualization-exception information area (see Section 25.5.6.2);

## 25.5.6 Virtualization Exceptions

A **virtualization exception** is a new processor exception. It uses vector 20 and is abbreviated #VE.

A virtualization exception can occur only in VMX non-root operation. Virtualization exceptions occur only with certain settings of certain VM-execution controls. Generally, these settings imply that certain conditions that would normally cause VM exits instead cause virtualization exceptions

In particular, the 1-setting of the “EPT-violation #VE” VM-execution control causes some EPT violations to generate virtualization exceptions instead of VM exits. Section 25.5.6.1 provides the details of how the processor determines whether an EPT violation causes a virtualization exception or a VM exit.

When the processor encounters a virtualization exception, it saves information about the exception to the virtualization-exception information area; see Section 25.5.6.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 25.5.6.3 for details.

### 25.5.6.1 Convertible EPT Violations

If the “EPT-violation #VE” VM-execution control is 0 (e.g., on processors that do not support this feature), EPT violations always cause VM exits. If instead the control is 1, certain EPT violations may be converted to cause virtualization exceptions instead; such EPT violations are **convertible**.

The values of certain EPT paging-structure entries determine which EPT violations are convertible. Specifically, bit 63 of certain EPT paging-structure entries may be defined to mean **suppress #VE**:

- If bits 2:0 of an EPT paging-structure entry are all 0, the entry is not present.<sup>2</sup> If the processor encounters such an entry while translating a guest-physical address, it causes an EPT violation. The EPT violation is convertible if and only if bit 63 of the entry is 0.

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

2. If the “mode-based execute control for EPT” VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 or bit 10 is 1.

- If an EPT paging-structure entry is present, the following cases apply:
  - If the value of the EPT paging-structure entry is not supported, the entry is **misconfigured**. If the processor encounters such an entry while translating a guest-physical address, it causes an EPT misconfiguration (not an EPT violation). EPT misconfigurations always cause VM exits.
  - If the value of the EPT paging-structure entry is supported, the following cases apply:
    - If bit 7 of the entry is 1, or if the entry is an EPT PTE, the entry maps a page. If the processor uses such an entry to translate a guest-physical address, and if an access to that address causes an EPT violation, the EPT violation is convertible if and only if bit 63 of the entry is 0.
    - If bit 7 of the entry is 0 and the entry is not an EPT PTE, the entry references another EPT paging structure. The processor does not use the value of bit 63 of the entry to determine whether any subsequent EPT violation is convertible.

If an access to a guest-physical address causes an EPT violation, bit 63 of exactly one of the EPT paging-structure entries used to translate that address is used to determine whether the EPT violation is convertible: either a entry that is not present (if the guest-physical address does not translate to a physical address) or an entry that maps a page (if it does).

A convertible EPT violation instead causes a virtualization exception if the following all hold:

- CR0.PE = 1;
- the logical processor is not in the process of delivering an event through the IDT; and
- the 32 bits at offset 4 in the virtualization-exception information area are all 0.

Delivery of virtualization exceptions writes the value FFFFFFFFH to offset 4 in the virtualization-exception information area (see Section 25.5.6.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

### 25.5.6.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 24.6.18). Table 25-1 enumerates the data saved and the format of the area.

**Table 25-1. Format of the Virtualization-Exception Information Area**

Byte Offset	Contents
0	The 32-bit value that would have been saved into the VMCS as an exit reason had a VM exit occurred instead of the virtualization exception. For EPT violations, this value is 48 (00000030H)
4	FFFFFFFFH
8	The 64-bit value that would have been saved into the VMCS as an exit qualification had a VM exit occurred instead of the virtualization exception
16	The 64-bit value that would have been saved into the VMCS as a guest-linear address had a VM exit occurred instead of the virtualization exception
24	The 64-bit value that would have been saved into the VMCS as a guest-physical address had a VM exit occurred instead of the virtualization exception
32	The current 16-bit value of the EPTP index VM-execution control (see Section 24.6.18 and Section 25.5.5.3)

### 25.5.6.3 Delivery of Virtualization Exceptions

After saving virtualization-exception information, the processor treats a virtualization exception as it does other exceptions:

- If bit 20 (#VE) is 1 in the exception bitmap in the VMCS, a virtualization exception causes a VM exit (see below). If the bit is 0, the virtualization exception is delivered using gate descriptor 20 in the IDT.

- Virtualization exceptions produce no error code. Delivery of a virtualization exception pushes no error code on the stack.
- With respect to double faults, virtualization exceptions have the same severity as page faults. If delivery of a virtualization exception encounters a nested fault that is either contributory or a page fault, a double fault (#DF) is generated. See Chapter 6, “Interrupt 8—Double Fault Exception (#DF)” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

It is not possible for a virtualization exception to be encountered while delivering another exception (see Section 25.5.6.1).

If a virtualization exception causes a VM exit directly (because bit 20 is 1 in the exception bitmap), information about the exception is saved normally in the VM-exit interruption information field in the VMCS (see Section 27.2.2). Specifically, the event is reported as a hardware exception with vector 20 and no error code. Bit 12 of the field (NMI unblocking due to IRET) is set normally.

If a virtualization exception causes a VM exit indirectly (because bit 20 is 0 in the exception bitmap and delivery of the exception generates an event that causes a VM exit), information about the exception is saved normally in the IDT-vectoring information field in the VMCS (see Section 27.2.3). Specifically, the event is reported as a hardware exception with vector 20 and no error code.

## 25.6 UNRESTRICTED GUESTS

The first processors to support VMX operation require CR0.PE and CR0.PG to be 1 in VMX operation (see Section 23.8). This restriction implies that guest software cannot be run in unpagged protected mode or in real-address mode. Later processors support a VM-execution control called “unrestricted guest”.<sup>1</sup> If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation. Such processors allow guest software to run in unpagged protected mode or in real-address mode. The following items describe the behavior of such software:

- The MOV CR0 instructions does not cause a general-protection exception simply because it would set either CR0.PE and CR0.PG to 0. See Section 25.3 for details.
- A logical processor treats the values of CR0.PE and CR0.PG in VMX non-root operation just as it does outside VMX operation. Thus, if CR0.PE = 0, the processor operates as it does normally in real-address mode (for example, it uses the 16-bit interrupt table to deliver interrupts and exceptions). If CR0.PG = 0, the processor operates as it does normally when paging is disabled.
- Processor operation is modified by the fact that the processor is in VMX non-root operation and by the settings of the VM-execution controls just as it is in protected mode or when paging is enabled. Instructions, interrupts, and exceptions that cause VM exits in protected mode or when paging is enabled also do so in real-address mode or when paging is disabled. The following examples should be noted:
  - If CR0.PG = 0, page faults do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, invalid-TSS exceptions do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, the following instructions cause invalid-opcode exceptions and do not cause VM exits: INVEPT, INVVPID, LLDT, LTR, SLDT, STR, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.
- If CR0.PG = 0, each linear address is passed directly to the EPT mechanism for translation to a physical address.<sup>2</sup> The guest memory type passed on to the EPT mechanism is WB (writeback).

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

2. As noted in Section 26.2.1.1, the “enable EPT” VM-execution control must be 1 if the “unrestricted guest” VM-execution control is 1.



## 15. Updates to Chapter 26, Volume 3C

Change bars show changes to Chapter 26 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes to this chapter: Updates to Section 26.5.1.1 "EPT Switching" and Section 26.6.3 "Delivery of Pending Debug Exceptions after VM Entry".

Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. VMLAUNCH can be used only with a VMCS whose launch state is clear and VMRESUME can be used only with a VMCS whose the launch state is launched. VMLAUNCH should be used for the first VM entry after VMCLEAR; VMRESUME should be used for subsequent VM entries with the same VMCS.

Each VM entry performs the following steps in the order indicated:

1. Basic checks are performed to ensure that VM entry can commence (Section 26.1).
2. The control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation and that the VMCS is correctly configured to support the next VM exit (Section 26.2).
3. The following may be performed in parallel or in any order (Section 26.3):
  - The guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures.
  - Processor state is loaded from the guest-state area and based on controls in the VMCS.
  - Address-range monitoring is cleared.
4. MSRs are loaded from the VM-entry MSR-load area (Section 26.4).
5. If VMLAUNCH is being executed, the launch state of the VMCS is set to “launched.”
6. An event may be injected in the guest context (Section 26.5).

Steps 1–4 above perform checks that may cause VM entry to fail. Such failures occur in one of the following three ways:

- Some of the checks in Section 26.1 may generate ordinary faults (for example, an invalid-opcode exception). Such faults are delivered normally.
- Some of the checks in Section 26.1 and all the checks in Section 26.2 cause control to pass to the instruction following the VM-entry instruction. The failure is indicated by setting RFLAGS.ZF<sup>1</sup> (if there is a current VMCS) or RFLAGS.CF (if there is no current VMCS). If there is a current VMCS, an error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 30 for the error numbers.
- The checks in Section 26.3 and Section 26.4 cause processor state to be loaded from the host-state area of the VMCS (as would be done on a VM exit). Information about the failure is stored in the VM-exit information fields. See Section 26.7 for details.

EFLAGS.TF = 1 causes a VM-entry instruction to generate a single-step debug exception only if failure of one of the checks in Section 26.1 and Section 26.2 causes control to pass to the following instruction. A VM-entry does not generate a single-step debug exception in any of the following cases: (1) the instruction generates a fault; (2) failure of one of the checks in Section 26.3 or in loading MSRs causes processor state to be loaded from the host-state area of the VMCS; or (3) the instruction passes all checks in Section 26.1, Section 26.2, and Section 26.3 and there is no failure in loading MSRs.

Section 34.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, code running in SMM returns using VM entries instead of the RSM instruction. A VM entry returns from SMM if it is executed in SMM and the “entry to SMM” VM-entry control is 0. VM entries that return from SMM differ from ordinary VM entries in ways that are detailed in Section 34.15.4.

---

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.



## 26.1 BASIC VM-ENTRY CHECKS

Before a VM entry commences, the current state of the logical processor is checked in the following order:

1. If the logical processor is in virtual-8086 mode or compatibility mode, an invalid-opcode exception is generated.
2. If the current privilege level (CPL) is not zero, a general-protection exception is generated.
3. If there is no current VMCS, RFLAGS.CF is set to 1 and control passes to the next instruction.
4. If there is a current VMCS but the current VMCS is a shadow VMCS (see Section 24.10), RFLAGS.CF is set to 1 and control passes to the next instruction.
5. If there is a current VMCS that is not a shadow VMCS, the following conditions are evaluated in order; any of these cause VM entry to fail:
  - a. if there is MOV-SS blocking (see Table 24-3)
  - b. if the VM entry is invoked by VMLAUNCH and the VMCS launch state is not clear
  - c. if the VM entry is invoked by VMRESUME and the VMCS launch state is not launched

If any of these checks fail, RFLAGS.ZF is set to 1 and control passes to the next instruction. An error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 30 for the error numbers.

## 26.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA

If the checks in Section 26.1 do not cause VM entry to fail, the control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation, that the VMCS is correctly configured to support the next VM exit, and that, after the next VM exit, the processor's state is consistent with the Intel 64 and IA-32 architectures.

VM entry fails if any of these checks fail. When such failures occur, control is passed to the next instruction, RFLAGS.ZF is set to 1 to indicate the failure, and the VM-instruction error field is loaded with an error number that indicates whether the failure was due to the controls or the host-state area (see Chapter 30).

These checks may be performed in any order. Thus, an indication by error number of one cause (for example, host state) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same VMCS. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The checks on the controls and the host-state area are presented in Section 26.2.1 through Section 26.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

### 26.2.1 Checks on VMX Controls

This section identifies VM-entry checks on the VMX control fields.

#### 26.2.1.1 VM-Execution Control Fields

VM entries perform the following checks on the VM-execution control fields:<sup>1</sup>

- Reserved bits in the pin-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.3.1).

---

1. If the "activate secondary controls" primary processor-based VM-execution control is 0, VM entry operates as if each secondary processor-based VM-execution control were 0.

- Reserved bits in the primary processor-based VM-execution controls must be set properly. Software may consult the VMX capability MSR to determine the proper settings (see Appendix A.3.2).
- If the “activate secondary controls” primary processor-based VM-execution control is 1, reserved bits in the secondary processor-based VM-execution controls must be cleared. Software may consult the VMX capability MSR to determine which bits are reserved (see Appendix A.3.3).  
If the “activate secondary controls” primary processor-based VM-execution control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the secondary processor-based VM-execution controls. The logical processor operates as if all the secondary processor-based VM-execution controls were 0.
- The CR3-target count must not be greater than 4. Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32\_VMX\_MISC to determine the number of values supported (see Appendix A.6).
- If the “use I/O bitmaps” VM-execution control is 1, bits 11:0 of each I/O-bitmap address must be 0. Neither address should set any bits beyond the processor’s physical-address width.<sup>1,2</sup>
- If the “use MSR bitmaps” VM-execution control is 1, bits 11:0 of the MSR-bitmap address must be 0. The address should not set any bits beyond the processor’s physical-address width.<sup>3</sup>
- If the “use TPR shadow” VM-execution control is 1, the virtual-APIC address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address should not set any bits beyond the processor’s physical-address width.<sup>4</sup>

If all of the above checks are satisfied and the “use TPR shadow” VM-execution control is 1, bytes 3:1 of VTPR (see Section 29.1.1) may be cleared (behavior may be implementation-specific).

The clearing of these bytes may occur even if the VM entry fails. This is true either if the failure causes control to pass to the instruction following the VM-entry instruction or if it causes processor state to be loaded from the host-state area of the VMCS.

- If the “use TPR shadow” VM-execution control is 1 and the “virtual-interrupt delivery” VM-execution control is 0, bits 31:4 of the TPR threshold VM-execution control field must be 0.<sup>5</sup>
- The following check is performed if the “use TPR shadow” VM-execution control is 1 and the “virtualize APIC accesses” and “virtual-interrupt delivery” VM-execution controls are both 0: the value of bits 3:0 of the TPR threshold VM-execution control field should not be greater than the value of bits 7:4 of VTPR (see Section 29.1.1).
- If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” VM-execution control must be 0.
- If the “virtual NMIs” VM-execution control is 0, the “NMI-window exiting” VM-execution control must be 0.
- If the “virtualize APIC-accesses” VM-execution control is 1, the APIC-access address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address should not set any bits beyond the processor’s physical-address width.<sup>6</sup>
- If the “use TPR shadow” VM-execution control is 0, the following VM-execution controls must also be 0: “virtualize x2APIC mode”, “APIC-register virtualization”, and “virtual-interrupt delivery”.<sup>7</sup>

---

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. If IA32\_VMX\_BASIC[48] is read as 1, these addresses must not set any bits in the range 63:32; see Appendix A.1.

3. If IA32\_VMX\_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

4. If IA32\_VMX\_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

5. “Virtual-interrupt delivery” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “virtual-interrupt delivery” VM-execution control were 0. See Section 24.6.2.

6. If IA32\_VMX\_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

7. “Virtualize x2APIC mode” and “APIC-register virtualization” are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if these controls were 0. See Section 24.6.2.

- If the “virtualize x2APIC mode” VM-execution control is 1, the “virtualize APIC accesses” VM-execution control must be 0.
- If the “virtual-interrupt delivery” VM-execution control is 1, the “external-interrupt exiting” VM-execution control must be 1.
- If the “process posted interrupts” VM-execution control is 1, the following must be true:<sup>1</sup>
  - The “virtual-interrupt delivery” VM-execution control is 1.
  - The “acknowledge interrupt on exit” VM-exit control is 1.
  - The posted-interrupt notification vector has a value in the range 0–255 (bits 15:8 are all 0).
  - Bits 5:0 of the posted-interrupt descriptor address are all 0.
  - The posted-interrupt descriptor address does not set any bits beyond the processor’s physical-address width.<sup>2</sup>
- If the “enable VPID” VM-execution control is 1, the value of the VPID VM-execution control field must not be 0000H.<sup>3</sup>
- If the “enable EPT” VM-execution control is 1, the EPTP VM-execution control field (see Table 24-8 in Section 24.6.11) must satisfy the following checks:<sup>4</sup>
  - The EPT memory type (bits 2:0) must be a value supported by the processor as indicated in the IA32\_VMX\_EPT\_VPID\_CAP MSR (see Appendix A.10).
  - Bits 5:3 (1 less than the EPT page-walk length) must be 3, indicating an EPT page-walk length of 4; see Section 28.2.2.
  - Bit 6 (enable bit for accessed and dirty flags for EPT) must be 0 if bit 21 of the IA32\_VMX\_EPT\_VPID\_CAP MSR (see Appendix A.10) is read as 0, indicating that the processor does not support accessed and dirty flags for EPT.
  - Reserved bits 11:7 and 63:N (where N is the processor’s physical-address width) must all be 0.
- If the “enable PML” VM-execution control is 1, the “enable EPT” VM-execution control must also be 1.<sup>5</sup> In addition, the PML address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address should not set any bits beyond the processor’s physical-address width.<sup>6</sup>
- If either the “unrestricted guest” VM-execution control or the “mode-based execute control for EPT” VM-execution control is 1, the “enable EPT” VM-execution control must also be 1.<sup>7</sup>
- If the “enable VM functions” processor-based VM-execution control is 1, reserved bits in the VM-function controls must be clear.<sup>8</sup> Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.11). In addition, the following check is performed based on the setting of bits in the VM-function controls (see Section 24.6.14):

---

1. “Process posted interrupts” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “process posted interrupts” VM-execution control were 0. See Section 24.6.2.

2. If IA32\_VMX\_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

3. “Enable VPID” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable VPID” VM-execution control were 0. See Section 24.6.2.

4. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.

5. “Enable PML” and “enable EPT” are both secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if both these controls were 0. See Section 24.6.2.

6. If IA32\_VMX\_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.

7. All these controls are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if all these controls were 0. See Section 24.6.2.

8. “Enable VM functions” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable VM functions” VM-execution control were 0. See Section 24.6.2.

- If “EPTP switching” VM-function control is 1, the “enable EPT” VM-execution control must also be 1. In addition, the EPTP-list address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address must not set any bits beyond the processor’s physical-address width.

If the “enable VM functions” processor-based VM-execution control is 0, no checks are performed on the VM-function controls.

- If the “VMCS shadowing” VM-execution control is 1, the VMREAD-bitmap and VMWRITE-bitmap addresses must each satisfy the following checks:<sup>1</sup>
  - Bits 11:0 of the address must be 0.
  - The address must not set any bits beyond the processor’s physical-address width.
- If the “EPT-violation #VE” VM-execution control is 1, the virtualization-exception information address must satisfy the following checks:<sup>2</sup>
  - Bits 11:0 of the address must be 0.
  - The address must not set any bits beyond the processor’s physical-address width.

### 26.2.1.2 VM-Exit Control Fields

VM entries perform the following checks on the VM-exit control fields.

- Reserved bits in the VM-exit controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.4).
- If the “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-exit control must also be 0.
- The following checks are performed for the VM-exit MSR-store address if the VM-exit MSR-store count field is non-zero:
  - The lower 4 bits of the VM-exit MSR-store address must be 0. The address should not set any bits beyond the processor’s physical-address width.<sup>3</sup>
  - The address of the last byte in the VM-exit MSR-store area should not set any bits beyond the processor’s physical-address width. The address of this last byte is VM-exit MSR-store address + (MSR count \* 16) – 1. (The arithmetic used for the computation uses more bits than the processor’s physical-address width.)

If IA32\_VMX\_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

- The following checks are performed for the VM-exit MSR-load address if the VM-exit MSR-load count field is non-zero:
  - The lower 4 bits of the VM-exit MSR-load address must be 0. The address should not set any bits beyond the processor’s physical-address width.
  - The address of the last byte in the VM-exit MSR-load area should not set any bits beyond the processor’s physical-address width. The address of this last byte is VM-exit MSR-load address + (MSR count \* 16) – 1. (The arithmetic used for the computation uses more bits than the processor’s physical-address width.)

If IA32\_VMX\_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

---

1. “VMCS shadowing” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “VMCS shadowing” VM-execution control were 0. See Section 24.6.2.

2. “EPT-violation #VE” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “EPT-violation #VE” VM-execution control were 0. See Section 24.6.2.

3. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

### 26.2.1.3 VM-Entry Control Fields

VM entries perform the following checks on the VM-entry control fields.

- Reserved bits in the VM-entry controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.5).
- Fields relevant to VM-entry event injection must be set properly. These fields are the VM-entry interruption-information field (see Table 24-13 in Section 24.8.3), the VM-entry exception error code, and the VM-entry instruction length. If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the following must hold:
  - The field's interruption type (bits 10:8) is not set to a reserved value. Value 1 is reserved on all logical processors; value 7 (other event) is reserved on logical processors that do not support the 1-setting of the "monitor trap flag" VM-execution control.
  - The field's vector (bits 7:0) is consistent with the interruption type:
    - If the interruption type is non-maskable interrupt (NMI), the vector is 2.
    - If the interruption type is hardware exception, the vector is at most 31.
    - If the interruption type is other event, the vector is 0 (pending MTF VM exit).
  - The field's deliver-error-code bit (bit 11) is 1 if and only if (1) either (a) the "unrestricted guest" VM-execution control is 0; or (b) bit 0 (corresponding to CR0.PE) is set in the CR0 field in the guest-state area; (2) the interruption type is hardware exception; and (3) the vector indicates an exception that would normally deliver an error code (8 = #DF; 10 = TS; 11 = #NP; 12 = #SS; 13 = #GP; 14 = #PF; or 17 = #AC).
  - Reserved bits in the field (30:12) are 0.
  - If the deliver-error-code bit (bit 11) is 1, bits 31:15 of the VM-entry exception error-code field are 0.
  - If the interruption type is software interrupt, software exception, or privileged software exception, the VM-entry instruction-length field is in the range 0–15. A VM-entry instruction length of 0 is allowed only if IA32\_VMX\_MISC[30] is read as 1; see Appendix A.6.
- The following checks are performed for the VM-entry MSR-load address if the VM-entry MSR-load count field is non-zero:
  - The lower 4 bits of the VM-entry MSR-load address must be 0. The address should not set any bits beyond the processor's physical-address width.<sup>1</sup>
  - The address of the last byte in the VM-entry MSR-load area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-entry MSR-load address + (MSR count \* 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

If IA32\_VMX\_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.
- If the processor is not in SMM, the "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls must be 0.
- The "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls cannot both be 1.

## 26.2.2 Checks on Host Control Registers and MSRs

The following checks are performed on fields in the host-state area that correspond to control registers and MSRs:

- The CR0 field must not set any bit to a value not supported in VMX operation (see Section 23.8).<sup>2</sup>
- The CR4 field must not set any bit to a value not supported in VMX operation (see Section 23.8).

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. The bits corresponding to CR0.NW (bit 29) and CR0.CD (bit 30) are never checked because the values of these bits are not changed by VM exit; see Section 27.5.1.

- On processors that support Intel 64 architecture, the CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width must be 0.<sup>1,2</sup>
- On processors that support Intel 64 architecture, the IA32\_SYSENTER\_ESP field and the IA32\_SYSENTER\_EIP field must each contain a canonical address.
- If the "load IA32\_PERF\_GLOBAL\_CTRL" VM-exit control is 1, bits reserved in the IA32\_PERF\_GLOBAL\_CTRL MSR must be 0 in the field for that register (see Figure 18-3).
- If the "load IA32\_PAT" VM-exit control is 1, the value of the field for the IA32\_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
- If the "load IA32\_EFER" VM-exit control is 1, bits reserved in the IA32\_EFER MSR must be 0 in the field for that register. In addition, the values of the LMA and LME bits in the field must each be that of the "host address-space size" VM-exit control.

### 26.2.3 Checks on Host Segment and Descriptor-Table Registers

The following checks are performed on fields in the host-state area that correspond to segment and descriptor-table registers:

- In the selector field for each of CS, SS, DS, ES, FS, GS and TR, the RPL (bits 1:0) and the TI flag (bit 2) must be 0.
- The selector fields for CS and TR cannot be 0000H.
- The selector field for SS cannot be 0000H if the "host address-space size" VM-exit control is 0.
- On processors that support Intel 64 architecture, the base-address fields for FS, GS, GDTR, IDTR, and TR must contain canonical addresses.

### 26.2.4 Checks Related to Address-Space Size

On processors that support Intel 64 architecture, the following checks related to address-space size are performed on VMX controls and fields in the host-state area:

- If the logical processor is outside IA-32e mode (if IA32\_EFER.LMA = 0) at the time of VM entry, the following must hold:
  - The "IA-32e mode guest" VM-entry control is 0.
  - The "host address-space size" VM-exit control is 0.
- If the logical processor is in IA-32e mode (if IA32\_EFER.LMA = 1) at the time of VM entry, the "host address-space size" VM-exit control must be 1.
- If the "host address-space size" VM-exit control is 0, the following must hold:
  - The "IA-32e mode guest" VM-entry control is 0.
  - Bit 17 of the CR4 field (corresponding to CR4.PCIDE) is 0.
  - Bits 63:32 in the RIP field is 0.
- If the "host address-space size" VM-exit control is 1, the following must hold:
  - Bit 5 of the CR4 field (corresponding to CR4.PAE) is 1.
  - The RIP field contains a canonical address.

On processors that do not support Intel 64 architecture, checks are performed to ensure that the "IA-32e mode guest" VM-entry control and the "host address-space size" VM-exit control are both 0.

- 
1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
  2. Bit 63 of the CR3 field in the host-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.

## 26.3 CHECKING AND LOADING GUEST STATE

If all checks on the VMX controls and the host-state area pass (see Section 26.2), the following operations take place concurrently: (1) the guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures; (2) processor state is loaded from the guest-state area or as specified by the VM-entry control fields; and (3) address-range monitoring is cleared.

Because the checking and the loading occur concurrently, a failure may be discovered only after some state has been loaded. For this reason, the logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 26.7.

### 26.3.1 Checks on the Guest State Area

This section describes checks performed on fields in the guest-state area. These checks may be performed in any order. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The following subsections reference fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

#### 26.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs

The following checks are performed on fields in the guest-state area corresponding to control registers, debug registers, and MSRs:

- The CR0 field must not set any bit to a value not supported in VMX operation (see Section 23.8). The following are exceptions:
  - Bit 0 (corresponding to CR0.PE) and bit 31 (PG) are not checked if the “unrestricted guest” VM-execution control is 1.<sup>1</sup>
  - Bit 29 (corresponding to CR0.NW) and bit 30 (CD) are never checked because the values of these bits are not changed by VM entry; see Section 26.3.2.1.
- If bit 31 in the CR0 field (corresponding to PG) is 1, bit 0 in that field (PE) must also be 1.<sup>2</sup>
- The CR4 field must not set any bit to a value not supported in VMX operation (see Section 23.8).
- If the “load debug controls” VM-entry control is 1, bits reserved in the IA32\_DEBUGCTL MSR must be 0 in the field for that register. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally.
- The following checks are performed on processors that support Intel 64 architecture:
  - If the “IA-32e mode guest” VM-entry control is 1, bit 31 in the CR0 field (corresponding to CR0.PG) and bit 5 in the CR4 field (corresponding to CR4.PAE) must each be 1.<sup>3</sup>
  - If the “IA-32e mode guest” VM-entry control is 0, bit 17 in the CR4 field (corresponding to CR4.PCIDE) must be 0.
  - The CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width are 0.<sup>4,5</sup>

---

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

2. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

3. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

4. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- If the “load debug controls” VM-entry control is 1, bits 63:32 in the DR7 field must be 0. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally (if they supported Intel 64 architecture).
- The IA32\_SYSENTER\_ESP field and the IA32\_SYSENTER\_EIP field must each contain a canonical address.
- If the “load IA32\_PERF\_GLOBAL\_CTRL” VM-entry control is 1, bits reserved in the IA32\_PERF\_GLOBAL\_CTRL MSR must be 0 in the field for that register (see Figure 18-3).
- If the “load IA32\_PAT” VM-entry control is 1, the value of the field for the IA32\_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
- If the “load IA32\_EFER” VM-entry control is 1, the following checks are performed on the field for the IA32\_EFER MSR:
  - Bits reserved in the IA32\_EFER MSR must be 0.
  - Bit 10 (corresponding to IA32\_EFER.LMA) must equal the value of the “IA-32e mode guest” VM-entry control. It must also be identical to bit 8 (LME) if bit 31 in the CR0 field (corresponding to CR0.PG) is 1.<sup>1</sup>
- If the “load IA32\_BNDCFGS” VM-entry control is 1, the following checks are performed on the field for the IA32\_BNDCFGS MSR:
  - Bits reserved in the IA32\_BNDCFGS MSR must be 0.
  - The linear address in bits 63:12 must be canonical.

### 26.3.1.2 Checks on Guest Segment Registers

This section specifies the checks on the fields for CS, SS, DS, ES, FS, GS, TR, and LDTR. The following terms are used in defining these checks:

- The guest will be **virtual-8086** if the VM flag (bit 17) is 1 in the RFLAGS field in the guest-state area.
- The guest will be **IA-32e mode** if the “IA-32e mode guest” VM-entry control is 1. (This is possible only on processors that support Intel 64 architecture.)
- Any one of these registers is said to be **usable** if the unusable bit (bit 16) is 0 in the access-rights field for that register.

The following are the checks on these fields:

- Selector fields.
  - TR. The TI flag (bit 2) must be 0.
  - LDTR. If LDTR is usable, the TI flag (bit 2) must be 0.
  - SS. If the guest will not be virtual-8086 and the “unrestricted guest” VM-execution control is 0, the RPL (bits 1:0) must equal the RPL of the selector field for CS.<sup>2</sup>
- Base-address fields.
  - CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the address must be the selector field shifted left 4 bits (multiplied by 16).
  - The following checks are performed on processors that support Intel 64 architecture:
    - TR, FS, GS. The address must be canonical.
    - LDTR. If LDTR is usable, the address must be canonical.
    - CS. Bits 63:32 of the address must be zero.

---

5. Bit 63 of the CR3 field in the guest-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.

1. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.



- SS, DS, ES. If the register is usable, bits 63:32 of the address must be zero.
- Limit fields for CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the field must be 0000FFFFH.
- Access-rights fields.
  - CS, SS, DS, ES, FS, GS.
    - If the guest will be virtual-8086, the field must be 000000F3H. This implies the following:
      - Bits 3:0 (Type) must be 3, indicating an expand-up read/write accessed data segment.
      - Bit 4 (S) must be 1.
      - Bits 6:5 (DPL) must be 3.
      - Bit 7 (P) must be 1.
      - Bits 11:8 (reserved), bit 12 (software available), bit 13 (reserved/L), bit 14 (D/B), bit 15 (G), bit 16 (unusable), and bits 31:17 (reserved) must all be 0.
    - If the guest will not be virtual-8086, the different sub-fields are considered separately:
      - Bits 3:0 (Type).
        - CS. The values allowed depend on the setting of the “unrestricted guest” VM-execution control:
          - If the control is 0, the Type must be 9, 11, 13, or 15 (accessed code segment).
          - If the control is 1, the Type must be either 3 (read/write accessed expand-up data segment) or one of 9, 11, 13, and 15 (accessed code segment).
        - SS. If SS is usable, the Type must be 3 or 7 (read/write, accessed data segment).
        - DS, ES, FS, GS. The following checks apply if the register is usable:
          - Bit 0 of the Type must be 1 (accessed).
          - If bit 3 of the Type is 1 (code segment), then bit 1 of the Type must be 1 (readable).
      - Bit 4 (S). If the register is CS or if the register is usable, S must be 1.
      - Bits 6:5 (DPL).
        - CS.
          - If the Type is 3 (read/write accessed expand-up data segment), the DPL must be 0. The Type can be 3 only if the “unrestricted guest” VM-execution control is 1.
          - If the Type is 9 or 11 (non-conforming code segment), the DPL must equal the DPL in the access-rights field for SS.
          - If the Type is 13 or 15 (conforming code segment), the DPL cannot be greater than the DPL in the access-rights field for SS.
        - SS.
          - If the “unrestricted guest” VM-execution control is 0, the DPL must equal the RPL from the selector field.
          - The DPL must be 0 either if the Type in the access-rights field for CS is 3 (read/write accessed expand-up data segment) or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.<sup>1</sup>
        - DS, ES, FS, GS. The DPL cannot be less than the RPL in the selector field if (1) the “unrestricted guest” VM-execution control is 0; (2) the register is usable; and (3) the Type in the access-rights field is in the range 0 – 11 (data segment or non-conforming code segment).
      - Bit 7 (P). If the register is CS or if the register is usable, P must be 1.

---

1. The following apply if either the “unrestricted guest” VM-execution control or bit 31 of the primary processor-based VM-execution controls is 0: (1) bit 0 in the CR0 field must be 1 if the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PE must be 1 in VMX operation; and (2) the Type in the access-rights field for CS cannot be 3.

- Bits 11:8 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
- Bit 14 (D/B). For CS, D/B must be 0 if the guest will be IA-32e mode and the L bit (bit 13) in the access-rights field is 1.
- Bit 15 (G). The following checks apply if the register is CS or if the register is usable:
  - If any bit in the limit field in the range 11:0 is 0, G must be 0.
  - If any bit in the limit field in the range 31:20 is 1, G must be 1.
- Bits 31:17 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
- TR. The different sub-fields are considered separately:
  - Bits 3:0 (Type).
    - If the guest will not be IA-32e mode, the Type must be 3 (16-bit busy TSS) or 11 (32-bit busy TSS).
    - If the guest will be IA-32e mode, the Type must be 11 (64-bit busy TSS).
  - Bit 4 (S). S must be 0.
  - Bit 7 (P). P must be 1.
  - Bits 11:8 (reserved). These bits must all be 0.
  - Bit 15 (G).
    - If any bit in the limit field in the range 11:0 is 0, G must be 0.
    - If any bit in the limit field in the range 31:20 is 1, G must be 1.
  - Bit 16 (Unusable). The unusable bit must be 0.
  - Bits 31:17 (reserved). These bits must all be 0.
- LDTR. The following checks on the different sub-fields apply only if LDTR is usable:
  - Bits 3:0 (Type). The Type must be 2 (LDT).
  - Bit 4 (S). S must be 0.
  - Bit 7 (P). P must be 1.
  - Bits 11:8 (reserved). These bits must all be 0.
  - Bit 15 (G).
    - If any bit in the limit field in the range 11:0 is 0, G must be 0.
    - If any bit in the limit field in the range 31:20 is 1, G must be 1.
  - Bits 31:17 (reserved). These bits must all be 0.

### 26.3.1.3 Checks on Guest Descriptor-Table Registers

The following checks are performed on the fields for GDTR and IDTR:

- On processors that support Intel 64 architecture, the base-address fields must contain canonical addresses.
- Bits 31:16 of each limit field must be 0.

### 26.3.1.4 Checks on Guest RIP and RFLAGS

The following checks are performed on fields in the guest-state area corresponding to RIP and RFLAGS:

- RIP. The following checks are performed on processors that support Intel 64 architecture:
  - Bits 63:32 must be 0 if the “IA-32e mode guest” VM-entry control is 0 or if the L bit (bit 13) in the access-rights field for CS is 0.
  - If the processor supports  $N < 64$  linear-address bits, bits 63:N must be identical if the “IA-32e mode guest” VM-entry control is 1 and the L bit in the access-rights field for CS is 1.<sup>1</sup> (No check applies if the processor supports 64 linear-address bits.)

- RFLAGS.
  - Reserved bits 63:22 (bits 31:22 on processors that do not support Intel 64 architecture), bit 15, bit 5 and bit 3 must be 0 in the field, and reserved bit 1 must be 1.
  - The VM flag (bit 17) must be 0 either if the “IA-32e mode guest” VM-entry control is 1 or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.<sup>1</sup>
  - The IF flag (RFLAGS[bit 9]) must be 1 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) is external interrupt.

### 26.3.1.5 Checks on Guest Non-Register State

The following checks are performed on fields in the guest-state area corresponding to non-register state:

- Activity state.
  - The activity-state field must contain a value in the range 0 – 3, indicating an activity state supported by the implementation (see Section 24.4.2). Future processors may include support for other activity states. Software should read the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine what activity states are supported.
  - The activity-state field must not indicate the HLT state if the DPL (bits 6:5) in the access-rights field for SS is not 0.<sup>2</sup>
  - The activity-state field must indicate the active state if the interruptibility-state field indicates blocking by either MOV-SS or by STI (if either bit 0 or bit 1 in that field is 1).
  - If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the interruption to be delivered (as defined by interruption type and vector) must not be one that would normally be blocked while a logical processor is in the activity state corresponding to the contents of the activity-state field. The following items enumerate the interruptions (as specified in the VM-entry interruption-information field) whose injection is allowed for the different activity states:
    - Active. Any interruption is allowed.
    - HLT. The only events allowed are the following:
      - Those with interruption type external interrupt or non-maskable interrupt (NMI).
      - Those with interruption type hardware exception and vector 1 (debug exception) or vector 18 (machine-check exception).
      - Those with interruption type other event and vector 0 (pending MTF VM exit).
 See Table 24-13 in Section 24.8.3 for details regarding the format of the VM-entry interruption-information field.
    - Shutdown. Only NMIs and machine-check exceptions are allowed.
    - Wait-for-SIPI. No interruptions are allowed.
  - The activity-state field must not indicate the wait-for-SIPI state if the “entry to SMM” VM-entry control is 1.
- Interruptibility state.
  - The reserved bits (bits 31:5) must be 0.
  - The field cannot indicate blocking by both STI and MOV SS (bits 0 and 1 cannot both be 1).
  - Bit 0 (blocking by STI) must be 0 if the IF flag (bit 9) is 0 in the RFLAGS field.

---

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

1. If the capability MSR IA32\_VMX\_CRO\_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. As noted in Section 24.4.1, SS.DPL corresponds to the logical processor’s current privilege level (CPL).

- Bit 0 (blocking by STI) and bit 1 (blocking by MOV-SS) must both be 0 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) in that field has value 0, indicating external interrupt.
- Bit 1 (blocking by MOV-SS) must be 0 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) in that field has value 2, indicating non-maskable interrupt (NMI).
- Bit 2 (blocking by SMI) must be 0 if the processor is not in SMM.
- Bit 2 (blocking by SMI) must be 1 if the “entry to SMM” VM-entry control is 1.
- A processor may require bit 0 (blocking by STI) to be 0 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) in that field has value 2, indicating NMI. Other processors may not make this requirement.
- Bit 3 (blocking by NMI) must be 0 if the “virtual NMIs” VM-execution control is 1, the valid bit (bit 31) in the VM-entry interruption-information field is 1, and the interruption type (bits 10:8) in that field has value 2 (indicating NMI).
- If bit 4 (enclave interruption) is 1, bit 1 (blocking by MOV-SS) must be 0 and the processor must support for SGX by enumerating CPUID.(EAX=07H,ECX=0):EBX.SGX[bit 2] as 1.

### NOTE

If the “virtual NMIs” VM-execution control is 0, there is no requirement that bit 3 be 0 if the valid bit in the VM-entry interruption-information field is 1 and the interruption type in that field has value 2.

- Pending debug exceptions.
  - Bits 11:4, bit 13, bit 15, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0.
  - The following checks are performed if any of the following holds: (1) the interruptibility-state field indicates blocking by STI (bit 0 in that field is 1); (2) the interruptibility-state field indicates blocking by MOV SS (bit 1 in that field is 1); or (3) the activity-state field indicates HLT:
    - Bit 14 (BS) must be 1 if the TF flag (bit 8) in the RFLAGS field is 1 and the BTF flag (bit 1) in the IA32\_DEBUGCTL field is 0.
    - Bit 14 (BS) must be 0 if the TF flag (bit 8) in the RFLAGS field is 0 or the BTF flag (bit 1) in the IA32\_DEBUGCTL field is 1.
  - The following checks are performed if bit 16 (RTM) is 1:
    - Bits 11:0, bits 15:13, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0; bit 12 must be 1.
    - The processor must support for RTM by enumerating CPUID.(EAX=07H,ECX=0):EBX[bit 11] as 1.
    - The interruptibility-state field must not indicate blocking by MOV SS (bit 1 in that field must be 0).
- VMCS link pointer. The following checks apply if the field contains a value other than FFFFFFFF\_FFFFFFFFH:
  - Bits 11:0 must be 0.
  - Bits beyond the processor’s physical-address width must be 0.<sup>1,2</sup>
  - The 4 bytes located in memory referenced by the value of the field (as a physical address) must satisfy the following:
    - Bits 30:0 must contain the processor’s VMCS revision identifier (see Section 24.2).<sup>3</sup>

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. If IA32\_VMX\_BASIC[48] is read as 1, this field must not set any bits in the range 63:32; see Appendix A.1.

3. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.

- Bit 31 must contain the setting of the “VMCS shadowing” VM-execution control.<sup>1</sup> This implies that the referenced VMCS is a shadow VMCS (see Section 24.10) if and only if the “VMCS shadowing” VM-execution control is 1.
- If the processor is not in SMM or the “entry to SMM” VM-entry control is 1, the field must not contain the current VMCS pointer.
- If the processor is in SMM and the “entry to SMM” VM-entry control is 0, the field must differ from the executive-VMCS pointer.

### 26.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32\_EFER.LME = 0, the logical processor uses **PAE paging** (see Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).<sup>2</sup> When PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs.

A VM entry is to a guest that uses PAE paging if (1) bit 31 (corresponding to CR0.PG) is set in the CR0 field in the guest-state area; (2) bit 5 (corresponding to CR4.PAE) is set in the CR4 field; and (3) the “IA-32e mode guest” VM-entry control is 0. Such a VM entry checks the validity of the PDPTEs:

- If the “enable EPT” VM-execution control is 0, VM entry checks the validity of the PDPTEs referenced by the CR3 field in the guest-state area if either (1) PAE paging was not in use before the VM entry; or (2) the value of CR3 is changing as a result of the VM entry. VM entry may check their validity even if neither (1) nor (2) hold.<sup>3</sup>
- If the “enable EPT” VM-execution control is 1, VM entry checks the validity of the PDPTE fields in the guest-state area (see Section 24.4.2).

A VM entry to a guest that does not use PAE paging does not check the validity of any PDPTEs.

A VM entry that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use.<sup>4</sup> If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), the VM entry fails.

## 26.3.2 Loading Guest State

Processor state is updated on VM entries in the following ways:

- Some state is loaded from the guest-state area.
- Some state is determined by VM-entry controls.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order and in parallel with the checking of VMCS contents (see Section 26.3.1).

The loading of guest state is detailed in Section 26.3.2.1 to Section 26.3.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

In addition to the state loading described in this section, VM entries may load MSRs from the VM-entry MSR-load area (see Section 26.4). This loading occurs only after the state loading described in this section and the checking of VMCS contents described in Section 26.3.1.

- 
1. “VMCS shadowing” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “VMCS shadowing” VM-execution control were 0. See Section 24.6.2.
  2. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine the number physical-address bits supported by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
  3. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.
  4. This implies that (1) bits 11:9 in each PDPTE are ignored; and (2) if bit 0 (present) is clear in one of the PDPTEs, bits 63:1 of that PDPTE are ignored.

### 26.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs

The following items describe how guest control registers, debug registers, and MSRs are loaded on VM entry:

- CR0 is loaded from the CR0 field with the exception of the following bits, which are never modified on VM entry: ET (bit 4); reserved bits 15:6, 17, and 28:19; NW (bit 29) and CD (bit 30).<sup>1</sup> The values of these bits in the CR0 field are ignored.
- CR3 and CR4 are loaded from the CR3 field and the CR4 field, respectively.
- If the “load debug controls” VM-entry control is 1, DR7 is loaded from the DR7 field with the exception that bit 12 and bits 15:14 are always 0 and bit 10 is always 1. The values of these bits in the DR7 field are ignored. The first processors to support the virtual-machine extensions supported only the 1-setting of the “load debug controls” VM-entry control and thus always loaded DR7 from the DR7 field.
- The following describes how certain MSRs are loaded using fields in the guest-state area:
  - If the “load debug controls” VM-entry control is 1, the IA32\_DEBUGCTL MSR is loaded from the IA32\_DEBUGCTL field. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always loaded the IA32\_DEBUGCTL MSR from the IA32\_DEBUGCTL field.
  - The IA32\_SYSENTER\_CS MSR is loaded from the IA32\_SYSENTER\_CS field. Since this field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  - The IA32\_SYSENTER\_ESP and IA32\_SYSENTER\_EIP MSRs are loaded from the IA32\_SYSENTER\_ESP field and the IA32\_SYSENTER\_EIP field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
  - The following are performed on processors that support Intel 64 architecture:
    - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 26.3.2.2).
    - If the “load IA32\_EFER” VM-entry control is 0, bits in the IA32\_EFER MSR are modified as follows:
      - IA32\_EFER.LMA is loaded with the setting of the “IA-32e mode guest” VM-entry control.
      - If CR0 is being loaded so that CR0.PG = 1, IA32\_EFER.LME is also loaded with the setting of the “IA-32e mode guest” VM-entry control.<sup>2</sup> Otherwise, IA32\_EFER.LME is unmodified.

See below for the case in which the “load IA32\_EFER” VM-entry control is 1

  - If the “load IA32\_PERF\_GLOBAL\_CTRL” VM-entry control is 1, the IA32\_PERF\_GLOBAL\_CTRL MSR is loaded from the IA32\_PERF\_GLOBAL\_CTRL field.
  - If the “load IA32\_PAT” VM-entry control is 1, the IA32\_PAT MSR is loaded from the IA32\_PAT field.
  - If the “load IA32\_EFER” VM-entry control is 1, the IA32\_EFER MSR is loaded from the IA32\_EFER field.
  - If the “load IA32\_BNDCFGS” VM-entry control is 1, the IA32\_BNDCFGS MSR is loaded from the IA32\_BNDCFGS field.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-entry MSR-load area. See Section 26.4.
- The SMBASE register is unmodified by all VM entries except those that return from SMM.

---

1. Bits 15:6, bit 17, and bit 28:19 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. Bits 15:6, bit 17, and bit 28:19 of CR0 are always 0 and CR0.ET is always 1.

2. If the capability MSR IA32\_VMX\_CRO\_FIXED0 reports that CR0.PG must be 1 in VMX operation, VM entry must be loading CR0 so that CR0.PG = 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

### 26.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers

For each of CS, SS, DS, ES, FS, GS, TR, and LDTR, fields are loaded from the guest-state area as follows:

- The unusable bit is loaded from the access-rights field. This bit can never be set for TR (see Section 26.3.1.2). If it is set for one of the other registers, the following apply:
  - For each of CS, SS, DS, ES, FS, and GS, uses of the segment cause faults (general-protection exception or stack-fault exception) outside 64-bit mode, just as they would had the segment been loaded using a null selector. This bit does not cause accesses to fault in 64-bit mode.
  - If this bit is set for LDTR, uses of LDTR cause general-protection exceptions in all modes, just as they would had LDTR been loaded using a null selector.

If this bit is clear for any of CS, SS, DS, ES, FS, GS, TR, and LDTR, a null selector value does not cause a fault (general-protection exception or stack-fault exception).
- TR. The selector, base, limit, and access-rights fields are loaded.
- CS.
  - The following fields are always loaded: selector, base address, limit, and (from the access-rights field) the L, D, and G bits.
  - For the other fields, the unusable bit of the access-rights field is consulted:
    - If the unusable bit is 0, all of the access-rights field is loaded.
    - If the unusable bit is 1, the remainder of CS access rights are undefined after VM entry.
- SS, DS, ES, FS, GS, and LDTR.
  - The selector fields are loaded.
  - For the other fields, the unusable bit of the corresponding access-rights field is consulted:
    - If the unusable bit is 0, the base-address, limit, and access-rights fields are loaded.
    - If the unusable bit is 1, the base address, the segment limit, and the remainder of the access rights are undefined after VM entry with the following exceptions:
      - Bits 3:0 of the base address for SS are cleared to 0.
      - SS.DPL is always loaded from the SS access-rights field. This will be the current privilege level (CPL) after the VM entry completes.
      - SS.B is always set to 1.
      - The base addresses for FS and GS are loaded from the corresponding fields in the VMCS. On processors that support Intel 64 architecture, the values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
      - On processors that support Intel 64 architecture, the base address for LDTR is set to an undefined but canonical value.
      - On processors that support Intel 64 architecture, bits 63:32 of the base addresses for SS, DS, and ES are cleared to 0.

GDTR and IDTR are loaded using the base and limit fields.

### 26.3.2.3 Loading Guest RIP, RSP, and RFLAGS

RSP, RIP, and RFLAGS are loaded from the RSP field, the RIP field, and the RFLAGS field, respectively. The following items regard the upper 32 bits of these fields on VM entries that are not to 64-bit mode:

- Bits 63:32 of RSP are undefined outside 64-bit mode. Thus, a logical processor may ignore the contents of bits 63:32 of the RSP field on VM entries that are not to 64-bit mode.
- As noted in Section 26.3.1.4, bits 63:32 of the RIP and RFLAGS fields must be 0 on VM entries that are not to 64-bit mode.

### 26.3.2.4 Loading Page-Directory-Pointer-Table Entries

As noted in Section 26.3.1.6, the logical processor uses PAE paging if  $CR0.PG = 1$ ,  $CR4.PAE = 1$ , and  $IA32\_EFER.LME = 0$ . A VM entry to a guest that uses PAE paging loads the PDPTEs into internal, non-architectural registers based on the setting of the “enable EPT” VM-execution control:

- If the control is 0, the PDPTEs are loaded from the page-directory-pointer table referenced by the physical address in the value of CR3 being loaded by the VM entry (see Section 26.3.2.1). The values loaded are treated as physical addresses in VMX non-root operation.
- If the control is 1, the PDPTEs are loaded from corresponding fields in the guest-state area (see Section 24.4.2). The values loaded are treated as guest-physical addresses in VMX non-root operation.

### 26.3.2.5 Updating Non-Register State

Section 28.3 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM entries invalidate cached mappings:

- If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP).
- VM entries are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

If the “virtual-interrupt delivery” VM-execution control is 1, VM entry loads the values of RVI and SVI from the guest interrupt-status field in the VMCS (see Section 24.4.2). After doing so, the logical processor first causes PPR virtualization (Section 29.1.3) and then evaluates pending virtual interrupts (Section 29.2.1).

If a virtual interrupt is recognized, it may be delivered in VMX non-root operation immediately after VM entry (including any specified event injection) completes; see Section 26.6.5. See Section 29.2.2 for details regarding the delivery of virtual interrupts.

### 26.3.3 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 8.10.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. VM entries clear any address-range monitoring that may be in effect.

## 26.4 LOADING MSRS

VM entries may load MSRs from the VM-entry MSR-load area (see Section 24.8.2). Specifically each entry in that area (up to the number specified in the VM-entry MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.<sup>1</sup>

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32\_FS\_BASE MSR) or C0000101 (the IA32\_GS\_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM entry did not commence in SMM. (IA32\_SMM\_MONITOR\_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM entries for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

1. Because attempts to modify the value of IA32\_EFER.LMA by WRMSR are ignored, attempts to modify it using the VM-entry MSR-load area are also ignored.



- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.<sup>1</sup>

The VM entry fails if processing fails for any entry. The logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 26.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM entry, the logical processor will not use any translations that were cached before the transition.

## 26.5 EVENT INJECTION

If the valid bit in the VM-entry interruption-information field (see Section 24.8.3) is 1, VM entry causes an event to be delivered (or made pending) after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.

- If the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception), the event is delivered as described in Section 26.5.1.
- If the interruption type in the field is 7 (other event) and the vector field is 0, an MTF VM exit is pending after VM entry. See Section 26.5.2.

### 26.5.1 Vectored-Event Injection

VM entry delivers an injected vectored event within the guest context established by VM entry. This means that delivery occurs after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.<sup>2</sup> The event is delivered using the vector in that field to select a descriptor in the IDT. Since event injection occurs after loading IDTR from the guest-state area, this is the guest IDT.

Section 26.5.1.1 provides details of vectored-event injection. In general, the event is delivered exactly as if it had been generated normally.

If event delivery encounters a nested exception (for example, a general-protection exception because the vector indicates a descriptor beyond the IDT limit), the exception bitmap is consulted using the vector of that exception:

- If the bit for the nested exception is 0, the nested exception is delivered normally. If the nested exception is benign, it is delivered through the IDT. If it is contributory or a page fault, a double fault may be generated, depending on the nature of the event whose delivery encountered the nested exception. See Chapter 6, “Interrupt 8—Double Fault Exception (#DF)” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.<sup>3</sup>
- If the bit for the nested exception is 1, a VM exit occurs. Section 26.5.1.2 details cases in which event injection causes a VM exit.

---

1. If CR0.PG = 1, WRMSR to the IA32\_EFER MSR causes a general-protection exception if it would modify the LME bit. If VM entry has established CR0.PG = 1, the IA32\_EFER MSR should not be included in the VM-entry MSR-load area for the purpose of modifying the LME bit.

2. This does not imply that injection of an exception or interrupt will cause a VM exit due to the settings of VM-execution control fields (such as the exception bitmap) that would cause a VM exit if the event had occurred in VMX non-root operation. In contrast, a nested exception encountered during event delivery may cause a VM exit; see Section 26.5.1.1.

3. Hardware exceptions with the following unused vectors are considered benign: 15 and 21–31. A hardware exception with vector 20 is considered benign unless the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control; in that case, it has the same severity as page faults.

### 26.5.1.1 Details of Vectored-Event Injection

The event-injection process is controlled by the contents of the VM-entry interruption information field (format given in Table 24-13), the VM-entry exception error-code field, and the VM-entry instruction-length field. The following items provide details of the process:

- The value pushed on the stack for RFLAGS is generally that which was loaded from the guest-state area. The value pushed for the RF flag is not modified based on the type of event being delivered. However, the pushed value of RFLAGS may be modified if a software interrupt is being injected into a guest that will be in virtual-8086 mode (see below). After RFLAGS is pushed on the stack, the value in the RFLAGS register is modified as is done normally when delivering an event through the IDT.
- The instruction pointer that is pushed on the stack depends on the type of event and whether nested exceptions occur during its delivery. The term **current guest RIP** refers to the value to be loaded from the guest-state area. The value pushed is determined as follows:<sup>1</sup>
  - If VM entry successfully injects (with no nested exception) an event with interruption type external interrupt, NMI, or hardware exception, the current guest RIP is pushed on the stack.
  - If VM entry successfully injects (with no nested exception) an event with interruption type software interrupt, privileged software exception, or software exception, the current guest RIP is incremented by the VM-entry instruction length before being pushed on the stack.
  - If VM entry encounters an exception while injecting an event and that exception does not cause a VM exit, the current guest RIP is pushed on the stack regardless of event type or VM-entry instruction length. If the encountered exception does cause a VM exit that saves RIP, the saved RIP is current guest RIP.
- If the deliver-error-code bit (bit 11) is set in the VM-entry interruption-information field, the contents of the VM-entry exception error-code field is pushed on the stack as an error code would be pushed during delivery of an exception.
- DR6, DR7, and the IA32\_DEBUGCTL MSR are not modified by event injection, even if the event has vector 1 (normal deliveries of debug exceptions, which have vector 1, do update these registers).
- If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode (RFLAGS.VM = 1), no general-protection exception can occur due to RFLAGS.IOPL < 3. A VM monitor should check RFLAGS.IOPL before injecting such an event and, if desired, inject a general-protection exception instead of a software interrupt.
- If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode with virtual-8086 mode extensions (RFLAGS.VM = CR4.VME = 1), event delivery is subject to VME-based interrupt redirection based on the software interrupt redirection bitmap in the task-state segment (TSS) as follows:
  - If bit *n* in the bitmap is clear (where *n* is the number of the software interrupt), the interrupt is directed to an 8086 program interrupt handler: the processor uses a 16-bit interrupt-vector table (IVT) located at linear address zero. If the value of RFLAGS.IOPL is less than 3, the following modifications are made to the value of RFLAGS that is pushed on the stack: IOPL is set to 3, and IF is set to the value of VIF.
  - If bit *n* in the bitmap is set (where *n* is the number of the software interrupt), the interrupt is directed to a protected-mode interrupt handler. (In other words, the injection is treated as described in the next item.) In this case, the software interrupt does not invoke such a handler if RFLAGS.IOPL < 3 (a general-protection exception occurs instead). However, as noted above, RFLAGS.IOPL cannot cause an injected software interrupt to cause such an exception. Thus, in this case, the injection invokes a protected-mode interrupt handler independent of the value of RFLAGS.IOPL.

Injection of events of other types are not subject to this redirection.

- If VM entry is injecting a software interrupt (not redirected as described above) or software exception, privilege checking is performed on the IDT descriptor being accessed as would be the case for executions of INT *n*, INT3, or INTO (the descriptor's DPL cannot be less than CPL). There is no checking of RFLAGS.IOPL, even if the guest will be in virtual-8086 mode. Failure of this check may lead to a nested exception. Injection of an event with interruption type external interrupt, NMI, hardware exception, and privileged software exception, or with interruption type software interrupt and being redirected as described above, do not perform these checks.

---

1. While these items refer to RIP, the width of the value pushed (16 bits, 32 bits, or 64 bits) is determined normally.

- If VM entry is injecting a non-maskable interrupt (NMI) and the “virtual NMIs” VM-execution control is 1, virtual-NMI blocking is in effect after VM entry.
- The transition causes a last-branch record to be logged if the LBR bit is set in the IA32\_DEBUGCTL MSR. This is true even for events such as debug exceptions, which normally clear the LBR bit before delivery.
- The last-exception record MSRs (LERs) may be updated based on the setting of the LBR bit in the IA32\_DEBUGCTL MSR. Events such as debug exceptions, which normally clear the LBR bit before they are delivered, and therefore do not normally update the LERs, may do so as part of VM-entry event injection.
- If injection of an event encounters a nested exception, the value of the EXT bit (bit 0) in any error code for that nested exception is determined as follows:
  - If event being injected has interruption type external interrupt, NMI, hardware exception, or privileged software exception and encounters a nested exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
  - If event being injected is a software interrupt or a software exception and encounters a nested exception, the error code for that exception clears the EXT bit.
  - If event delivery encounters a nested exception and delivery of that exception encounters another exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
  - If a double fault is produced, the error code for the double fault is 0000H (the EXT bit is clear).

### 26.5.1.2 VM Exits During Event Injection

An event being injected never causes a VM exit directly regardless of the settings of the VM-execution controls. For example, setting the “NMI exiting” VM-execution control to 1 does not cause a VM exit due to injection of an NMI.

However, the event-delivery process may lead to a VM exit:

- If the vector in the VM-entry interruption-information field identifies a task gate in the IDT, the attempted task switch may cause a VM exit just as it would had the injected event occurred during normal execution in VMX non-root operation (see Section 25.4.2).
- If event delivery encounters a nested exception, a VM exit may occur depending on the contents of the exception bitmap (see Section 25.2).
- If event delivery generates a double-fault exception (due to a nested exception); the logical processor encounters another nested exception while attempting to call the double-fault handler; and that exception does not cause a VM exit due to the exception bitmap; then a VM exit occurs due to triple fault (see Section 25.2).
- If event delivery injects a double-fault exception and encounters a nested exception that does not cause a VM exit due to the exception bitmap, then a VM exit occurs due to triple fault (see Section 25.2).
- If the “virtualize APIC accesses” VM-execution control is 1 and event delivery generates an access to the APIC-access page, that access is treated as described in Section 29.4 and may cause a VM exit.<sup>1</sup>

If the event-delivery process does cause a VM exit, the processor state before the VM exit is determined just as it would be had the injected event occurred during normal execution in VMX non-root operation. If the injected event directly accesses a task gate that cause a VM exit or if the first nested exception encountered causes a VM exit, information about the injected event is saved in the IDT-vectoring information field (see Section 27.2.3).

### 26.5.1.3 Event Injection for VM Entries to Real-Address Mode

If VM entry is loading CR0.PE with 0, any injected vectored event is delivered as would normally be done in real-address mode.<sup>2</sup> Specifically, VM entry uses the vector provided in the VM-entry interruption-information field to select a 4-byte entry from an interrupt-vector table at the linear address in IDTR.base. Further details are provided in Section 15.1.4 in Volume 3A of the *IA-32 Intel® Architecture Software Developer’s Manual*.

1. “Virtualize APIC accesses” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “virtualize APIC accesses” VM-execution control were 0. See Section 24.6.2.
2. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PE must be 1 in VMX operation, VM entry must be loading CR0.PE with 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

Because bit 11 (deliver error code) in the VM-entry interruption-information field must be 0 if CR0.PE will be 0 after VM entry (see Section 26.2.1.3), vectored events injected with CR0.PE = 0 do not push an error code on the stack. This is consistent with event delivery in real-address mode.

If event delivery encounters a fault (due to a violation of IDTR.limit or of SS.limit), the fault is treated as if it had occurred during event delivery in VMX non-root operation. Such a fault may lead to a VM exit as discussed in Section 26.5.1.2.

## 26.5.2 Injection of Pending MTF VM Exits

If the interruption type in the VM-entry interruption-information field is 7 (other event) and the vector field is 0, VM entry causes an MTF VM exit to be pending on the instruction boundary following VM entry. This is the case even if the “monitor trap flag” VM-execution control is 0. See Section 25.5.2 for the treatment of pending MTF VM exits.

## 26.6 SPECIAL FEATURES OF VM ENTRY

This section details a variety of features of VM entry. It uses the following terminology: a VM entry is **vectoring** if the valid bit (bit 31) of the VM-entry interruption information field is 1 and the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception).

### 26.6.1 Interruptibility State

The interruptibility-state field in the guest-state area (see Table 24-3) contains bits that control blocking by STI, blocking by MOV SS, and blocking by NMI. This field impacts event blocking after VM entry as follows:

- If the VM entry is vectoring, there is no blocking by STI or by MOV SS following the VM entry, regardless of the contents of the interruptibility-state field.
- If the VM entry is not vectoring, the following apply:
  - Events are blocked by STI if and only if bit 0 in the interruptibility-state field is 1. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry; see Section 26.6.3).
  - Events are blocked by MOV SS if and only if bit 1 in the interruptibility-state field is 1. This may affect the treatment of pending debug exceptions; see Section 26.6.3. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry).
- The blocking of non-maskable interrupts (NMIs) is determined as follows:
  - If the “virtual NMIs” VM-execution control is 0, NMIs are blocked if and only if bit 3 (blocking by NMI) in the interruptibility-state field is 1. If the “NMI exiting” VM-execution control is 0, execution of the IRET instruction removes this blocking (even if the instruction generates a fault). If the “NMI exiting” control is 1, IRET does not affect this blocking.
  - The following items describe the use of bit 3 (blocking by NMI) in the interruptibility-state field if the “virtual NMIs” VM-execution control is 1:
    - The bit’s value does not affect the blocking of NMIs after VM entry. NMIs are not blocked in VMX non-root operation (except for ordinary blocking for other reasons, such as by the MOV SS instruction, the wait-for-SIPI state, etc.)
    - The bit’s value determines whether there is virtual-NMI blocking after VM entry. If the bit is 1, virtual-NMI blocking is in effect after VM entry. If the bit is 0, there is no virtual-NMI blocking after VM entry unless the VM entry is injecting an NMI (see Section 26.5.1.1). Execution of IRET removes virtual-NMI blocking (even if the instruction generates a fault).

If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” control must be 0; see Section 26.2.1.1.

- Blocking of system-management interrupts (SMIs) is determined as follows:
  - If the VM entry was not executed in system-management mode (SMM), SMI blocking is unchanged by VM entry.
  - If the VM entry was executed in SMM, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.

## 26.6.2 Activity State

The activity-state field in the guest-state area controls whether, after VM entry, the logical processor is active or in one of the inactive states identified in Section 24.4.2. The use of this field is determined as follows:

- If the VM entry is vectoring, the logical processor is in the active state after VM entry. While the consistency checks described in Section 26.3.1.5 on the activity-state field do apply in this case, the contents of the activity-state field do not determine the activity state after VM entry.
- If the VM entry is not vectoring, the logical processor ends VM entry in the activity state specified in the guest-state area. If VM entry ends with the logical processor in an inactive activity state, the VM entry generates any special bus cycle that is normally generated when that activity state is entered from the active state. If VM entry would end with the logical processor in the shutdown state and the logical processor is in SMX operation,<sup>1</sup> an Intel<sup>®</sup> TXT shutdown condition occurs. The error code used is 0000H, indicating “legacy shutdown.” See *Intel<sup>®</sup> Trusted Execution Technology Preliminary Architecture Specification*.
- Some activity states unconditionally block certain events. The following blocking is in effect after any VM entry that puts the processor in the indicated state:
  - The active state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the active state and in VMX non-root operation are discarded and do not cause VM exits.
  - The HLT state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the HLT state and in VMX non-root operation are discarded and do not cause VM exits.
  - The shutdown state blocks external interrupts and SIPIs. External interrupts that arrive while a logical processor is in the shutdown state and in VMX non-root operation do not cause VM exits even if the “external-interrupt exiting” VM-execution control is 1. SIPIs that arrive while a logical processor is in the shutdown state and in VMX non-root operation are discarded and do not cause VM exits.
  - The wait-for-SIPI state blocks external interrupts, non-maskable interrupts (NMIs), INIT signals, and system-management interrupts (SMIs). Such events do not cause VM exits if they arrive while a logical processor is in the wait-for-SIPI state and in VMX non-root operation.

## 26.6.3 Delivery of Pending Debug Exceptions after VM Entry

The pending debug exceptions field in the guest-state area indicates whether there are debug exceptions that have not yet been delivered (see Section 24.4.2). This section describes how these are treated on VM entry.

There are no pending debug exceptions after VM entry if any of the following are true:

- The VM entry is vectoring with one of the following interruption types: external interrupt, non-maskable interrupt (NMI), hardware exception, or privileged software exception.
- The interruptibility-state field does not indicate blocking by MOV SS and the VM entry is vectoring with either of the following interruption type: software interrupt or software exception.
- The VM entry is not vectoring and the activity-state field indicates either shutdown or wait-for-SIPI.

If none of the above hold, the pending debug exceptions field specifies the debug exceptions that are pending for the guest. There are **valid pending debug exceptions** if either the BS bit (bit 14) or the enable-breakpoint bit (bit 12) is 1. If there are valid pending debug exceptions, they are handled as follows:

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

- If the VM entry is not vectoring, the pending debug exceptions are treated as they would had they been encountered normally in guest execution:
  - If the logical processor is not blocking such exceptions (the interruptibility-state field indicates no blocking by MOV SS), a debug exception is delivered after VM entry (see below).
  - If the logical processor is blocking such exceptions (due to blocking by MOV SS), the pending debug exceptions are held pending or lost as would normally be the case.
- If the VM entry is vectoring (with interruption type software interrupt or software exception and with blocking by MOV SS), the following items apply:
  - For injection of a software interrupt or of a software exception with vector 3 (#BP) or vector 4 (#OF) — or a privileged software exception with vector 1 (#DB) — the pending debug exceptions are treated as they would had they been encountered normally in guest execution if the corresponding instruction (INT1, INT3, or INTO) were executed after a MOV SS that encountered a debug trap.
  - For injection of a software exception with a vector other than 3 and 4, the pending debug exceptions may be lost or they may be delivered after injection (see below).

If there are no valid pending debug exceptions (as defined above), no pending debug exceptions are delivered after VM entry.

If a pending debug exception is delivered after VM entry, it has the priority of “traps on the previous instruction” (see Section 6.9 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Thus, INIT signals and system-management interrupts (SMIs) take priority of such an exception, as do VM exits induced by the TPR threshold (see Section 26.6.7) and pending MTF VM exits (see Section 26.6.8). The exception takes priority over any pending non-maskable interrupt (NMI) or external interrupt and also over VM exits due to the 1-settings of the “interrupt-window exiting” and “NMI-window exiting” VM-execution controls.

A pending debug exception delivered after VM entry causes a VM exit if the bit 1 (#DB) is 1 in the exception bitmap. If it does not cause a VM exit, it updates DR6 normally.

## 26.6.4 VMX-Preemption Timer

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

It is possible for the VMX-preemption timer to expire during VM entry (e.g., if the value in the VMX-preemption timer-value field is zero). If this happens (and if the VM entry was not to the wait-for-SIPI state), a VM exit occurs with its normal priority after any event injection and before execution of any instruction following VM entry. For example, any pending debug exceptions established by VM entry (see Section 26.6.3) take priority over a timer-induced VM exit. (The timer-induced VM exit will occur after delivery of the debug exception, unless that exception or its delivery causes a different VM exit.)

See Section 25.5.1 for details of the operation of the VMX-preemption timer in VMX non-root operation, including the blocking and priority of the VM exits that it causes.

## 26.6.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery

If “interrupt-window exiting” VM-execution control is 1, an open interrupt window may cause a VM exit immediately after VM entry (see Section 25.2 for details). If the “interrupt-window exiting” VM-execution control is 0 but the “virtual-interrupt delivery” VM-execution control is 1, a virtual interrupt may be delivered immediately after VM entry (see Section 26.3.2.5 and Section 29.2.1).

The following items detail the treatment of these events:

- These events occur after any event injection specified for VM entry.
- Non-maskable interrupts (NMIs) and higher priority events take priority over these events. These events take priority over external interrupts and lower priority events.
- These events wake the logical processor if it just entered the HLT state because of a VM entry (see Section 26.6.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

## 26.6.6 NMI-Window Exiting

The “NMI-window exiting” VM-execution control may cause a VM exit to occur immediately after VM entry (see Section 25.2 for details).

The following items detail the treatment of these VM exits:

- These VM exits follow event injection if such injection is specified for VM entry.
- Debug-trap exceptions (see Section 26.6.3) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.
- VM exits caused by this control wake the logical processor if it just entered either the HLT state or the shutdown state because of a VM entry (see Section 26.6.2). They do not occur if the logical processor just entered the wait-for-SIPI state.

## 26.6.7 VM Exits Induced by the TPR Threshold

If the “use TPR shadow” and “virtualize APIC accesses” VM-execution controls are both 1 and the “virtual-interrupt delivery” VM-execution control is 0, a VM exit occurs immediately after VM entry if the value of bits 3:0 of the TPR threshold VM-execution control field is greater than the value of bits 7:4 of VTPR (see Section 29.1.1).<sup>1</sup>

The following items detail the treatment of these VM exits:

- The VM exits are not blocked if RFLAGS.IF = 0 or by the setting of bits in the interruptibility-state field in guest-state area.
- The VM exits follow event injection if such injection is specified for VM entry.
- VM exits caused by this control take priority over system-management interrupts (SMIs), INIT signals, and lower priority events. They thus have priority over the VM exits described in Section 26.6.5, Section 26.6.6, and Section 26.6.8, as well as any interrupts or debug exceptions that may be pending at the time of VM entry.
- These VM exits wake the logical processor if it just entered the HLT state as part of a VM entry (see Section 26.6.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state. If such a VM exit is suppressed because the processor just entered the shutdown state, it occurs after the delivery of any event that cause the logical processor to leave the shutdown state while remaining in VMX non-root operation (e.g., due to an NMI that occurs while the “NMI-exiting” VM-execution control is 0).
- The basic exit reason is “TPR below threshold.”

## 26.6.8 Pending MTF VM Exits

As noted in Section 26.5.2, VM entry may cause an MTF VM exit to be pending immediately after VM entry. The following items detail the treatment of these VM exits:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these VM exits. These VM exits take priority over debug-trap exceptions and lower priority events.
- These VM exits wake the logical processor if it just entered the HLT state because of a VM entry (see Section 26.6.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

## 26.6.9 VM Entries and Advanced Debugging Features

VM entries are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

---

1. “Virtualize APIC accesses” and “virtual-interrupt delivery” are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if these controls were 0. See Section 24.6.2.

## 26.7 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE

VM-entry failures due to the checks identified in Section 26.3.1 and failures during the MSR loading identified in Section 26.4 are treated differently from those that occur earlier in VM entry. In these cases, the following steps take place:

1. Information about the VM-entry failure is recorded in the VM-exit information fields:
  - Exit reason.
    - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM-entry failure. The following numbers are used:
      33. VM-entry failure due to invalid guest state. A VM entry failed one of the checks identified in Section 26.3.1.
      34. VM-entry failure due to MSR loading. A VM entry failed in an attempt to load MSRs (see Section 26.4).
      41. VM-entry failure due to machine-check event. A machine-check event occurred during VM entry (see Section 26.8).
    - Bit 31 is set to 1 to indicate a VM-entry failure.
    - The remainder of the field (bits 30:16) is cleared.
  - Exit qualification. This field is set based on the exit reason.
    - VM-entry failure due to invalid guest state. In most cases, the exit qualification is cleared to 0. The following non-zero values are used in the cases indicated:
      1. Not used.
      2. Failure was due to a problem loading the PDPTes (see Section 26.3.1.6).
      3. Failure was due to an attempt to inject a non-maskable interrupt (NMI) into a guest that is blocking events through the STI blocking bit in the interruptibility-state field. Such failures are implementation-specific (see Section 26.3.1.5).
      4. Failure was due to an invalid VMCS link pointer (see Section 26.3.1.5).

VM-entry checks on guest-state fields may be performed in any order. Thus, an indication by exit qualification of one cause does not imply that there are not also other errors. Different processors may give different exit qualifications for the same VMCS.
    - VM-entry failure due to MSR loading. The exit qualification is loaded to indicate which entry in the VM-entry MSR-load area caused the problem (1 for the first entry, 2 for the second, etc.).
  - All other VM-exit information fields are unmodified.
2. Processor state is loaded as would be done on a VM exit (see Section 27.5). If this results in  $[CR4.PAE \& CR0.PG \& \sim IA32\_EFER.LMA] = 1$ , page-directory-pointer-table entries (PDPTes) may be checked and loaded (see Section 27.5.4).
3. The state of blocking by NMI is what it was before VM entry.
4. MSRs are loaded as specified in the VM-exit MSR-load area (see Section 27.6).

Although this process resembles that of a VM exit, many steps taken during a VM exit do not occur for these VM-entry failures:

- Most VM-exit information fields are not updated (see step 1 above).
- The valid bit in the VM-entry interruption-information field is not cleared.
- The guest-state area is not modified.
- No MSRs are saved into the VM-exit MSR-store area.

## 26.8 MACHINE-CHECK EVENTS DURING VM ENTRY

If a machine-check event occurs during a VM entry, one of the following occurs:



- The machine-check event is handled as if it occurred before the VM entry:
  - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:<sup>1</sup>
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating “unrecoverable machine-check condition.”
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If CR4.MCE = 1, a machine-check exception (#MC) is delivered through the IDT.
- The machine-check event is handled after VM entry completes:
  - If the VM entry ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If the VM entry ends with CR4.MCE = 1, a machine-check exception (#MC) is generated:
    - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
    - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- A VM-entry failure occurs as described in Section 26.7. The basic exit reason is 41, for “VM-entry failure due to machine-check event.”

The first option is not used if the machine-check event occurs after any guest state has been loaded. The second option is used only if VM entry is able to load all guest state.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

## 16. Updates to Chapter 27, Volume 3C

Change bars show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

---

Changes to this chapter: Updates to Section 27.1 "Architectural State Before a VM Exit", Section 27.2.2 "Information for VM Exits Due to Vectored Events", Section 27.2.3 "Information for VM Exits During Event Delivery", Section 27.2.4 "Information for VM Exits Due to Instruction Execution", and Section 27.3.3 "Saving RIP, RSP, and RFLAGS".

VM exits occur in response to certain instructions and events in VMX non-root operation as detailed in Section 25.1 through Section 25.2. VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified as described in Section 27.2.
2. Processor state is saved in the guest-state area (Section 27.3).
3. MSRs may be saved in the VM-exit MSR-store area (Section 27.4). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.
4. The following may be performed in parallel and in any order (Section 27.5):
  - Processor state is loaded based in part on the host-state area and some VM-exit controls. This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM. See Section 34.15.6 for information on how processor state is loaded by such VM exits.
  - Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 27.6). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.

VM exits are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

Section 27.1 clarifies the nature of the architectural state before a VM exit begins. The steps described above are detailed in Section 27.2 through Section 27.6.

Section 34.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, ordinary transitions to SMM are replaced by VM exits to a separate SMM monitor. Called **SMM VM exits**, these are caused by the arrival of an SMI or the execution of VMCALL in VMX root operation. SMM VM exits differ from other VM exits in ways that are detailed in Section 34.15.2.

## 27.1 ARCHITECTURAL STATE BEFORE A VM EXIT

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT. Note the following:

- An exception causes a VM exit **directly** if the bit corresponding to that exception is set in the exception bitmap. A non-maskable interrupt (NMI) causes a VM exit directly if the “NMI exiting” VM-execution control is 1. An external interrupt causes a VM exit directly if the “external-interrupt exiting” VM-execution control is 1. A start-up IPI (SIPI) that arrives while a logical processor is in the wait-for-SIPI activity state causes a VM exit directly. INIT signals that arrive while the processor is not in the wait-for-SIPI activity state cause VM exits directly.
- An exception, NMI, external interrupt, or software interrupt causes a VM exit **indirectly** if it does not do so directly but delivery of the event causes a nested exception, double fault, task switch, APIC access (see Section 27.4), EPT violation, EPT misconfiguration, or page-modification log-full event that causes a VM exit.
- An event **results** in a VM exit if it causes a VM exit (directly or indirectly).

The following bullets detail when architectural state is and is not updated in response to VM exits:

- If an event causes a VM exit directly, it does not update architectural state as it would have if it had it not caused the VM exit:
  - A debug exception does not update DR6, DR7.GD, or IA32\_DEBUGCTL.LBR. (Information about the nature of the debug exception is saved in the exit qualification field.)
  - A page fault does not update CR2. (The linear address causing the page fault is saved in the exit-qualification field.)
  - An NMI causes subsequent NMIs to be blocked, but only after the VM exit completes.

- An external interrupt does not acknowledge the interrupt controller and the interrupt remains pending, unless the “acknowledge interrupt on exit” VM-exit control is 1. In such a case, the interrupt controller is acknowledged and the interrupt is no longer pending.
  - The flags L0 – L3 in DR7 (bit 0, bit 2, bit 4, and bit 6) are not cleared when a task switch causes a VM exit.
  - If a task switch causes a VM exit, none of the following are modified by the task switch: old task-state segment (TSS); new TSS; old TSS descriptor; new TSS descriptor; RFLAGS.NT<sup>1</sup>; or the TR register.
  - No last-exception record is made if the event that would do so directly causes a VM exit.
  - If a machine-check exception causes a VM exit directly, this does not prevent machine-check MSR from being updated. These are updated by the machine-check event itself and not the resulting machine-check exception.
  - If the logical processor is in an inactive state (see Section 24.4.2) and not executing instructions, some events may be blocked but others may return the logical processor to the active state. Unblocked events may cause VM exits.<sup>2</sup> If an unblocked event causes a VM exit directly, a return to the active state occurs only after the VM exit completes.<sup>3</sup> The VM exit generates any special bus cycle that is normally generated when the active state is entered from that activity state.
- MTF VM exits (see Section 25.5.2 and Section 26.6.8) are not blocked in the HLT activity state. If an MTF VM exit occurs in the HLT activity state, the logical processor returns to the active state only after the VM exit completes. MTF VM exits are blocked the shutdown state and the wait-for-SIPI state.
- If an event causes a VM exit indirectly, the event does update architectural state:
    - A debug exception updates DR6, DR7, and the IA32\_DEBUGCTL MSR. No debug exceptions are considered pending.
    - A page fault updates CR2.
    - An NMI causes subsequent NMIs to be blocked before the VM exit commences.
    - An external interrupt acknowledges the interrupt controller and the interrupt is no longer pending.
    - If the logical processor had been in an inactive state, it enters the active state and, before the VM exit commences, generates any special bus cycle that is normally generated when the active state is entered from that activity state.
    - There is no blocking by STI or by MOV SS when the VM exit commences.
    - Processor state that is normally updated as part of delivery through the IDT (CS, RIP, SS, RSP, RFLAGS) is not modified. However, the incomplete delivery of the event may write to the stack.
    - The treatment of last-exception records is implementation dependent:
      - Some processors make a last-exception record when beginning the delivery of an event through the IDT (before it can encounter a nested exception). Such processors perform this update even if the event encounters a nested exception that causes a VM exit (including the case where nested exceptions lead to a triple fault).
      - Other processors delay making a last-exception record until event delivery has reached some event handler successfully (perhaps after one or more nested exceptions). Such processors do not update the last-exception record if a VM exit or triple fault occurs before an event handler is reached.
  - If the “virtual NMIs” VM-execution control is 1, VM entry injects an NMI, and delivery of the NMI causes a nested exception, double fault, task switch, or APIC access that causes a VM exit, virtual-NMI blocking is in effect before the VM exit commences.

---

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

2. If a VM exit takes the processor from an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.

3. An exception is made if the logical processor had been inactive due to execution of MWAIT; in this case, it is considered to have become active before the VM exit.

- If a VM exit results from a fault, EPT violation, EPT misconfiguration, or page-modification log-full event is encountered during execution of IRET and the “NMI exiting” VM-execution control is 0, any blocking by NMI is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.1 and Section 27.2.2.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, or page-modification log-full event is encountered during execution of IRET and the “virtual NMIs” VM-execution control is 1, virtual-NMI blocking is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.1 and Section 27.2.2.
- Suppose that a VM exit is caused directly by an x87 FPU Floating-Point Error (#MF) or by any of the following events if the event was unblocked due to (and given priority over) an x87 FPU Floating-Point Error: an INIT signal, an external interrupt, an NMI, an SMI; or a machine-check exception. In these cases, there is no blocking by STI or by MOV SS when the VM exit commences.
- Normally, a last-branch record may be made when an event is delivered through the IDT. However, if such an event results in a VM exit before delivery is complete, no last-branch record is made.
- If machine-check exception results in a VM exit, processor state is suspect and may result in suspect state being saved to the guest-state area. A VM monitor should consult the RIPV and EIPV bits in the IA32\_MCG\_STATUS MSR before resuming a guest that caused a VM exit resulting from a machine-check exception.
- If a VM exit results from a fault, APIC access (see Section 29.4), EPT violation, EPT misconfiguration, or page-modification log-full event is encountered while executing an instruction, data breakpoints due to that instruction may have been recognized and information about them may be saved in the pending debug exceptions field (unless the VM exit clears that field; see Section 27.3.4).
- The following VM exits are considered to happen after an instruction is executed:
  - VM exits resulting from debug traps (single-step, I/O breakpoints, and data breakpoints).
  - VM exits resulting from debug exceptions (data breakpoints) whose recognition was delayed by blocking by MOV SS.
  - VM exits resulting from some machine-check exceptions.
  - Trap-like VM exits due to execution of MOV to CR8 when the “CR8-load exiting” VM-execution control is 0 and the “use TPR shadow” VM-execution control is 1 (see Section 29.3). (Such VM exits can occur only from 64-bit mode and thus only on processors that support Intel 64 architecture.)
  - Trap-like VM exits due to execution of WRMSR when the “use MSR bitmaps” VM-execution control is 1; the value of ECX is in the range 800H–8FFH; and the bit corresponding to the ECX value in write bitmap for low MSRs is 0; and the “virtualize x2APIC mode” VM-execution control is 1. See Section 29.5.
  - VM exits caused by APIC-write emulation (see Section 29.4.3.2) that result from APIC accesses as part of instruction execution.

For these VM exits, the instruction’s modifications to architectural state complete before the VM exit occurs. Such modifications include those to the logical processor’s interruptibility state (see Table 24-3). If there had been blocking by MOV SS, POP SS, or STI before the instruction executed, such blocking is no longer in effect.

A VM exit that occurs in enclave mode sets bit 27 of the exit-reason field and bit 4 of the guest interruptibility-state field. Before such a VM exit is delivered, an Asynchronous Enclave Exit (AEX) occurs (see Chapter 39, “Enclave Exiting Events”). An AEX modifies architectural state (Section 39.3). In particular, the processor establishes the following architectural state as indicated:

- The following bits in RFLAGS are cleared: CF, PF, AF, ZF, SF, OF, and RF.
- FS and GS are restored to the values they had prior to the most recent enclave entry.
- RIP is loaded with the AEP of interrupted enclave thread.
- RSP is loaded from the URSP field in the enclave’s state-save area (SSA).

## 27.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS

VM exits begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields. Section 27.2.1 to Section 27.2.4 detail the use of these fields.

In addition to updating the VM-exit information fields, the valid bit (bit 31) is cleared in the VM-entry interruption-information field. If bit 5 of the IA32\_VMX\_MISC MSR (index 485H) is read as 1 (see Appendix A.6), the value of IA32\_EFER.LMA is stored into the “IA-32e mode guest” VM-entry control.<sup>1</sup>

### 27.2.1 Basic VM-Exit Information

Section 24.9.1 defines the basic VM-exit information fields. The following items detail their use.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM exit. Appendix C lists the numbers used and their meaning.
  - Bit 27 of this field is set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode. A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1).
  - The remainder of the field (bits 31:28 and bits 26:16) is cleared to 0 (certain SMM VM exits may set some of these bits; see Section 34.15.2.3).<sup>2</sup>
- **Exit qualification.** This field is saved for VM exits due to the following causes: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); system-management interrupts (SMIs) that arrive immediately after the retirement of I/O instructions; task switches; INVEPT; INVLPG; INVPCID; INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; MWAIT; accesses to the APIC-access page (see Section 29.4); EPT violations; EOI virtualization (see Section 29.1.4); APIC-write emulation (see Section 29.4.3.3); and page-modification log full (see Section 28.2.5). For all other VM exits, this field is cleared. The following items provide details:
  - For a debug exception, the exit qualification contains information about the debug exception. The information has the format given in Table 27-1.

**Table 27-1. Exit Qualification for Debug Exceptions**

Bit Position(s)	Contents
3:0	B3 - B0. When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if its corresponding enabling bit in DR7 is not set.
12:4	Reserved (cleared to 0).
13	BD. When set, this bit indicates that the cause of the debug exception is “debug register access detected.”
14	BS. When set, this bit indicates that the cause of the debug exception is either the execution of a single instruction (if RFLAGS.TF = 1 and IA32_DEBUGCTL.BTF = 0) or a taken branch (if RFLAGS.TF = DEBUGCTL.BTF = 1).
63:15	Reserved (cleared to 0). Bits 63:32 exist only on processors that support Intel 64 architecture.

1. Bit 5 of the IA32\_VMX\_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.  
 2. Bit 31 of this field is set on certain VM-entry failures; see Section 26.7.

- For a page-fault exception, the exit qualification contains the linear address that caused the page fault. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

- For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.
- For a task switch, the exit qualification contains details about the task switch, encoded as shown in Table 27-2.
- For INVLPG, the exit qualification contains the linear-address operand of the instruction.
  - On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
  - If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

**Table 27-2. Exit Qualification for Task Switch**

Bit Position(s)	Contents
15:0	Selector of task-state segment (TSS) to which the guest attempted to switch
29:16	Reserved (cleared to 0)
31:30	Source of task switch initiation: 0: CALL instruction 1: IRET instruction 2: JMP instruction 3: Task gate in IDT
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

- For INVEPT, INVPCID, INVVPID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, and XSAVES, the exit qualification receives the value of the instruction's displacement field, which is sign-extended to 64 bits if necessary (32 bits on processors that do not support Intel 64 architecture). If the instruction has no displacement (for example, has a register operand), zero is stored into the exit qualification.

On processors that support Intel 64 architecture, an exception is made for RIP-relative addressing (used only in 64-bit mode). Such addressing causes an instruction to use an address that is the sum of the displacement field and the value of RIP that references the following instruction. In this case, the exit qualification is loaded with the sum of the displacement field and the appropriate RIP value.

In all cases, bits of this field beyond the instruction's address size are undefined. For example, suppose that the address-size field in the VM-exit instruction-information field (see Section 24.9.4 and Section 27.2.4) reports an  $n$ -bit address size. Then bits 63: $n$  (bits 31: $n$  on processors that do not support Intel 64 architecture) of the instruction displacement are undefined.

- For a control-register access, the exit qualification contains information about the access and has the format given in Table 27-3.
- For MOV DR, the exit qualification contains information about the instruction and has the format given in Table 27-4.
- For an I/O instruction, the exit qualification contains information about the instruction and has the format given in Table 27-5.

- For MWAIT, the exit qualification contains a value that indicates whether address-range monitoring hardware was armed. The exit qualification is set either to 0 (if address-range monitoring hardware is not armed) or to 1 (if address-range monitoring hardware is armed).
- For an APIC-access VM exit resulting from a linear access or a guest-physical access to the APIC-access page (see Section 29.4), the exit qualification contains information about the access and has the format given in Table 27-6.<sup>1</sup>

If the access to the APIC-access page occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

Such a VM exit that set bits 15:12 of the exit qualification to 0000b (data read during instruction execution) or 0001b (data write during instruction execution) set bit 12—which distinguishes data read from data write—to that which would have been stored in bit 1—W/R—of the page-fault error code had the access caused a page fault instead of an APIC-access VM exit. This implies the following:

- For an APIC-access VM exit caused by the CLFLUSH and CLFLUSHOPT instructions, the access type is “data read during instruction execution.”
- For an APIC-access VM exit caused by the ENTER instruction, the access type is “data write during instruction execution.”

**Table 27-3. Exit Qualification for Control-Register Accesses**

Bit Positions	Contents
3:0	Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.
5:4	Access type: 0 = MOV to CR 1 = MOV from CR 2 = CLTS 3 = LMSW
6	LMSW operand type: 0 = register 1 = memory  For CLTS and MOV CR, cleared to 0
7	Reserved (cleared to 0)
11:8	For MOV CR, the general-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture)  For CLTS and LMSW, cleared to 0
15:12	Reserved (cleared to 0)

1. The exit qualification is undefined if the access was part of the logging of a branch record or a processor-event-based-sampling (PEBS) record to the DS save area. It is recommended that software configure the paging structures so that no address in the DS save area translates to an address on the APIC-access page.



**Table 27-3. Exit Qualification for Control-Register Accesses (Contd.)**

Bit Positions	Contents
31:16	For LMSW, the LMSW source data For CLTS and MOV CR, cleared to 0
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

- For an APIC-access VM exit caused by the MASKMOVQ instruction or the MASKMOVDQU instruction, the access type is “data write during instruction execution.”
- For an APIC-access VM exit caused by the MONITOR instruction, the access type is “data read during instruction execution.”

Such a VM exit stores 1 for bit 31 for IDT-vectoring information field (see Section 27.2.3) if and only if it sets bits 15:12 of the exit qualification to 0011b (linear access during event delivery) or 1010b (guest-physical access during event delivery).

See Section 29.4.4 for further discussion of these instructions and APIC-access VM exits.

For APIC-access VM exits resulting from physical accesses to the APIC-access page (see Section 29.4.6), the exit qualification is undefined.

- For an EPT violation, the exit qualification contains information about the access causing the EPT violation and has the format given in Table 27-7.

As noted in that table, the format and meaning of the exit qualification depends on the setting of the “mode-based execute control for EPT” VM-execution control and whether the processor supports advanced VM-exit information for EPT violations.<sup>1</sup>

An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations.

**Table 27-4. Exit Qualification for MOV DR**

Bit Position(s)	Contents
2:0	Number of debug register
3	Reserved (cleared to 0)
4	Direction of access (0 = MOV to DR; 1 = MOV from DR)
7:5	Reserved (cleared to 0)
11:8	General-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8 - 15 = R8 - R15, respectively
63:12	Reserved (cleared to 0)

1. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP (see Appendix A.10).

**Table 27-5. Exit Qualification for I/O Instructions**

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte  Other values not used
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)
15:7	Reserved (cleared to 0)
31:16	Port number (as specified in DX or in an immediate operand)
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

Bit 12 is undefined in any of the following cases:

- If the “NMI exiting” VM-execution control is 1 and the “virtual NMIs” VM-execution control is 0.
- If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).

Otherwise, bit 12 is defined as follows:

- If the “virtual NMIs” VM-execution control is 0, the EPT violation was caused by a memory access as part of execution of the IRET instruction, and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.

**Table 27-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses**

Bit Position(s)	Contents
11:0	<ul style="list-style-type: none"> <li>▪ If the APIC-access VM exit is due to a linear access, the offset of access within the APIC page.</li> <li>▪ Undefined if the APIC-access VM exit is due a guest-physical access</li> </ul>
15:12	Access type: 0 = linear access for a data read during instruction execution 1 = linear access for a data write during instruction execution 2 = linear access for an instruction fetch 3 = linear access (read or write) during event delivery 10 = guest-physical access during event delivery 15 = guest-physical access for an instruction fetch or during instruction execution  Other values not used
63:16	Reserved (cleared to 0). Bits 63:32 exist only on processors that support Intel 64 architecture.

- If the “virtual NMIs” VM-execution control is 1, the EPT violation was caused by a memory access as part of execution of the IRET instruction, and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
  - For all other relevant VM exits, bit 12 is cleared to 0.
- For VM exits caused as part of EOI virtualization (Section 29.1.4), bits 7:0 of the exit qualification are set to vector of the virtual interrupt that was dismissed by the EOI virtualization. Bits above bit 7 are cleared.
  - For APIC-write VM exits (Section 29.4.3.3), bits 11:0 of the exit qualification are set to the page offset of the write access that caused the VM exit.<sup>1</sup> Bits above bit 11 are cleared.
  - For a VM exit due to a page-modification log-full event (Section 28.2.5), only bit 12 of the exit qualification is defined, and only in some cases. It is undefined in the following cases:
    - If the “NMI exiting” VM-execution control is 1 and the “virtual NMIs” VM-execution control is 0.
    - If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).

Otherwise, it is defined as follows:

- If the “virtual NMIs” VM-execution control is 0, the page-modification log-full event was caused by a memory access as part of execution of the IRET instruction, and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.
- If the “virtual NMIs” VM-execution control is 1, the page-modification log-full event was caused by a memory access as part of execution of the IRET instruction, and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
- For all other relevant VM exits, bit 12 is cleared to 0.

For these VM exits, all bits other than bit 12 are undefined.

- **Guest-linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:
  - VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
  - VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

**Table 27-7. Exit Qualification for EPT Violations**

Bit Position(s)	Contents
0	Set if the access causing the EPT violation was a data read. <sup>1</sup>
1	Set if the access causing the EPT violation was a data write. <sup>1</sup>
2	Set if the access causing the EPT violation was an instruction fetch.
3	The logical-AND of bit 0 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was readable). <sup>2</sup>
4	The logical-AND of bit 1 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was writeable).

1. Execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit; the exit qualification for such an APIC-write VM exit is 3FOH.

Table 27-7. Exit Qualification for EPT Violations (Contd.)

Bit Position(s)	Contents
5	The logical-AND of bit 2 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. If the “mode-based execute control for EPT” VM-execution control is 0, this indicates whether the guest-physical address was executable. If that control is 1, this indicates whether the guest-physical address was executable for supervisor-mode linear addresses.
6	If the “mode-based execute control” VM-execution control is 0, the value of this bit is undefined. If that control is 1, this bit is the logical-AND of bit 10 in the EPT paging-structures entries used to translate the guest-physical address of the access causing the EPT violation. In this case, it indicates whether the guest-physical address was executable for user-mode linear addresses.
7	Set if the guest linear-address field is valid. The guest linear-address field is valid for all EPT violations except those resulting from an attempt to load the guest PDPTes as part of the execution of the MOV CR instruction.
8	If bit 7 is 1: <ul style="list-style-type: none"> <li>▪ Set if the access causing the EPT violation is to a guest-physical address that is the translation of a linear address.</li> <li>▪ Clear if the access causing the EPT violation is to a paging-structure entry as part of a page walk or the update of an accessed or dirty bit.</li> </ul> Reserved if bit 7 is 0 (cleared to 0).
9	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, <sup>3</sup> this bit is 0 if the linear address is a supervisor-mode linear address and 1 if it is a user-mode linear address. (If CRO.PG = 0, the translation of every linear address is a user-mode linear address and thus this bit will be 1.) Otherwise, this bit is undefined.
10	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, <sup>3</sup> this bit is 0 if paging translates the linear address to a read-only page and 1 if it translates to a read/write page. (If CRO.PG = 0, every linear address is read/write and thus this bit will be 1.) Otherwise, this bit is undefined.
11	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations, <sup>3</sup> this bit is 0 if paging translates the linear address to an executable page and 1 if it translates to an execute-disable page. (If CRO.PG = 0, CR4.PAE = 0, or IA32_EFER.NXE = 0, every linear address is executable and thus this bit will be 0.) Otherwise, this bit is undefined.
12	NMI unblocking due to IRET
63:13	Reserved (cleared to 0).

**NOTES:**

1. If accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations (see Section 28.2.3.2). If such an access causes an EPT violation, the processor sets both bit 0 and bit 1 of the exit qualification.
2. Bits 5:3 are cleared to 0 if any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present (see Section 28.2.2).
3. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32\_VMX\_EPT\_VPID\_CAP (see Appendix A.10).
  - VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 27-7; these are all EPT violations except those resulting from an attempt to load the PDPTes as of execution of the MOV CR instruction). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the EPT violation occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

- For all other VM exits, the field is undefined.
- **Guest-physical address.** For a VM exit due to an EPT violation or an EPT misconfiguration, this field receives the guest-physical address that caused the EPT violation or EPT misconfiguration. For all other VM exits, the field is undefined.

If the EPT violation or EPT misconfiguration occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

## 27.2.2 Information for VM Exits Due to Vectored Events

Section 24.9.2 defines fields containing information for VM exits due to the following events: exceptions (including those generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs).<sup>1</sup> Such VM exits include those that occur on an attempt at a task switch that causes an exception before generating the VM exit due to the task switch that causes the VM exit.

The following items detail the use of these fields:

- **VM-exit interruption information** (format given in Table 24-15). The following items detail how this field is established for VM exits due to these events:
  - For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the vector.
  - Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 5 (privileged software exception), or 6 (software exception). Hardware exceptions comprise all exceptions except the following:
    - Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
    - Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

- Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).<sup>2</sup> If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).
- Bit 12 is undefined in any of the following cases:
  - If the “NMI exiting” VM-execution control is 1 and the “virtual NMIs” VM-execution control is 0.
  - If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).
  - If the VM exit is due to a double fault (the interruption type is hardware exception and the vector is 8).

Otherwise, bit 12 is defined as follows:

- If the “virtual NMIs” VM-execution control is 0, the VM exit is due to a fault on the IRET instruction (other than a debug exception for an instruction breakpoint), and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for *n*.

2. If the capability MSR IA32\_VMX\_CRO\_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

- If the “virtual NMIs” VM-execution control is 1, the VM exit is due to a fault on the IRET instruction (other than a debug exception for an instruction breakpoint), and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
  - For all other relevant VM exits, bit 12 is cleared to 0.<sup>1</sup>
- Bits 30:13 are always set to 0.
  - Bit 31 is always set to 1.

For other VM exits (including those due to external interrupts when the “acknowledge interrupt on exit” VM-exit control is 0), the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- VM-exit interruption error code.
  - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the VM-exit interruption-information field, this field receives the error code that would have been pushed on the stack had the event causing the VM exit been delivered normally through the IDT. The EXT bit is set in this field exactly when it would be set normally. For exceptions that occur during the delivery of double fault (if the IDT-vectoring information field indicates a double fault), the EXT bit is set to 1, assuming that (1) that the exception would produce an error code normally (if not incident to double-fault delivery) and (2) that the error code uses the EXT bit (not for page faults, which use a different format).
  - For other VM exits, the value of this field is undefined.

### 27.2.3 Information for VM Exits During Event Delivery

Section 24.9.3 defined fields containing information for VM exits that occur while delivering an event through the IDT and as a result of any of the following cases:<sup>2</sup>

- A fault occurs during event delivery and causes a VM exit (because the bit associated with the fault is set to 1 in the exception bitmap).
- A task switch is invoked through a task gate in the IDT. The VM exit occurs due to the task switch only after the initial checks of the task switch pass (see Section 25.4.2).
- Event delivery causes an APIC-access VM exit (see Section 29.4).
- An EPT violation, EPT misconfiguration, or page-modification log-full event that occurs during event delivery.

These fields are used for VM exits that occur during delivery of events injected as part of VM entry (see Section 26.5.1.2).

A VM exit is not considered to occur during event delivery in any of the following circumstances:

- The original event causes the VM exit directly (for example, because the original event is a non-maskable interrupt (NMI) and the “NMI exiting” VM-execution control is 1).
- The original event results in a double-fault exception that causes the VM exit directly.
- The VM exit occurred as a result of fetching the first instruction of the handler invoked by the event delivery.
- The VM exit is caused by a triple fault.

The following items detail the use of these fields:

- IDT-vectoring information (format given in Table 24-16). The following items detail how this field is established for VM exits that occur during event delivery:
  - If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the vector.

---

1. The conditions imply that, if the “NMI exiting” VM-execution control is 0 or the “virtual NMIs” VM-execution control is 1, bit 12 is always cleared to 0 by VM exits due to debug exceptions.

2. This includes the case in which a VM exit occurs while delivering a software interrupt (INT *n*) through the 16-bit IVT (interrupt vector table) that is used in virtual-8086 mode with virtual-machine extensions (if RFLAGS.VM = CR4.VME = 1).

- Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).

Hardware exceptions comprise all exceptions except the following:<sup>1</sup>

- Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
- Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

- Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).<sup>2</sup> If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).
- Bit 12 is undefined.
- Bits 30:13 are always set to 0.
- Bit 31 is always set to 1.

For other VM exits, the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- IDT-vectoring error code.
  - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the IDT-vectoring information field, this field receives the error code that would have been pushed on the stack by the event that was being delivered through the IDT at the time of the VM exit. The EXT bit is set in this field when it would be set normally.
  - For other VM exits, the value of this field is undefined.

## 27.2.4 Information for VM Exits Due to Instruction Execution

Section 24.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

- **VM-exit instruction length.** This field is used in the following cases:
  - For fault-like VM exits due to attempts to execute one of the following instructions that cause VM exits unconditionally (see Section 25.1.2) or based on the settings of VM-execution controls (see Section 25.1.3): CLTS, CPUID, ENCLS, GETSEC, HLT, IN, INS, INVD, INVEPT, INVLPG, INVPCID, INVVPID, LGDT, LIDT, LLDT, LMSW, LTR, MONITOR, MOV CR, MOV DR, MWAIT, OUT, OUTS, PAUSE, RDMSR, RDPMSR, RDRAND, RDSEED, RDTSC, RDTSCP, RSM, SGDT, SIDT, SLDT, STR, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, WBINVD, WRMSR, XRSTORS, XSETBV, and XSAVES.<sup>3</sup>

- 
1. In the following items, INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for *n*.
  2. If the capability MSR IA32\_VMX\_CRO\_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
  3. This item applies only to fault-like VM exits. It does not apply to trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1 or to those following executions of the WRMSR instruction when the “virtualize x2APIC mode” VM-execution control is 1.

- For VM exits due to software exceptions (those generated by executions of INT3 or INTO) or privileged software exceptions (those generated by executions of INT1).
- For VM exits due to faults encountered during delivery of a software interrupt, privileged software exception, or software exception.
- For VM exits due to attempts to effect a task switch via instruction execution. These are VM exits that produce an exit reason indicating task switch and either of the following:
  - An exit qualification indicating execution of CALL, IRET, or JMP instruction.
  - An exit qualification indicating a task gate in the IDT and an IDT-vectoring information field indicating that the task gate was encountered during delivery of a software interrupt, privileged software exception, or software exception.
- For APIC-access VM exits and for VM exits caused by EPT violations and page-modification log-full events encountered during delivery of a software interrupt, privileged software exception, or software exception.<sup>1</sup>
- For VM exits due to executions of VMFUNC that fail because one of the following is true:
  - EAX indicates a VM function that is not enabled (the bit at position EAX is 0 in the VM-function controls; see Section 25.5.5.2).
  - EAX = 0 and either ECX ≥ 512 or the value of ECX selects an invalid tentative EPTP value (see Section 25.5.5.3).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

The cases of VM exits encountered during delivery of a software interrupt, privileged software exception, or software exception include those encountered during delivery of events injected as part of VM entry (see Section 26.5.1.2). If the original event was injected as part of VM entry, this field receives the value of the VM-entry instruction length.

All VM exits other than those listed in the above items leave this field undefined.

If the VM exit occurred in enclave mode, this field is cleared (none of the previous items apply).

**Table 27-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS**

Bit Position(s)	Content
6:0	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
14:10	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for VM exits due to execution of INS.
31:18	Undefined.

- **VM-exit instruction information.** For VM exits due to attempts to execute INS, INVEPT, INVPCID, INVVPID, LIDT, LGDT, LLDT, LTR, OUTS, RDRAND, RDSEED, SIDT, SGDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST,

1. The VM-exit instruction-length field is not defined following APIC-access VM exits resulting from physical accesses (see Section 29.4.6) even if encountered during delivery of a software interrupt, privileged software exception, or software exception.



VMREAD, VMWRITE, VMXON, XRSTORS, or XSAVES, this field receives information about the instruction that caused the VM exit. The format of the field depends on the identity of the instruction causing the VM exit:

- For VM exits due to attempts to execute INS or OUTS, the field has the format is given in Table 27-8.<sup>1</sup>
- For VM exits due to attempts to execute INVEPT, INVPCID, or INVVPID, the field has the format is given in Table 27-9.
- For VM exits due to attempts to execute LIDT, LGDT, SIDT, or SGDT, the field has the format is given in Table 27-10.
- For VM exits due to attempts to execute LLDT, LTR, SLDT, or STR, the field has the format is given in Table 27-11.
- For VM exits due to attempts to execute RDRAND or RDSEED, the field has the format is given in Table 27-12.
- For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, or XSAVES, the field has the format is given in Table 27-13.
- For VM exits due to attempts to execute VMREAD or VMWRITE, the field has the format is given in Table 27-14.

For all other VM exits, the field is undefined, unless the VM exit occurred in enclave mode, in which case the field is cleared.

- **I/O RCX, I/O RSI, I/O RDI, I/O RIP.** These fields are undefined except for SMM VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions. See Section 34.15.2.3. Note that, if the VM exit occurred in enclave mode, these fields are all cleared.

**Table 27-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID**

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.

1. The format of the field was undefined for these VM exits on the first processors to support the virtual-machine extensions. Software can determine whether the format specified in Table 27-8 is used by consulting the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1).

**Table 27-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVPID (Contd.)**

Bit Position(s)	Content
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for memory instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Reg2 (same encoding as IndexReg above)

**Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT**

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
11	Operand size: 0: 16-bit 1: 32-bit Undefined for VM exits from 64-bit mode.
14:12	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.

**Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT (Contd.)**

Bit Position(s)	Content
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
29:28	Instruction identity: 0: SGDT 1: SIDT 2: LGDT 3: LIDT
31:30	Undefined.

**Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR**

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).

**Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR (Contd.)**

Bit Position(s)	Content
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS  Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
29:28	Instruction identity: 0: SLDT 1: STR 2: LLDT 3: LTR
31:30	Undefined.

**Table 27-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED**

Bit Position(s)	Content
2:0	Undefined.
6:3	Destination register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture)
10:7	Undefined.
12:11	Operand size: 0: 16-bit 1: 32-bit 2: 64-bit  The value 3 is not used.

**Table 27-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED**

Bit Position(s)	Content
31:13	Undefined.

**Table 27-13. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES**

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8-15 represent R8-R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Undefined.

**Table 27-14. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE**

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
31:28	Reg2 (same encoding as Reg1 above)

## 27.3 SAVING GUEST STATE

Each field in the guest-state area of the VMCS (see Section 24.4) is written with the corresponding component of processor state. On processors that support Intel 64 architecture, the full values of each natural-width field (see Section 24.11.2) is saved regardless of the mode of the logical processor before and after the VM exit.

In general, the state saved is that which was in the logical processor at the time the VM exit commences. See Section 27.1 for a discussion of which architectural updates occur at that time.

Section 27.3.1 through Section 27.3.4 provide details for how certain components of processor state are saved. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

### 27.3.1 Saving Control Registers, Debug Registers, and MSRs

Contents of certain control registers, debug registers, and MSRs is saved as follows:

- The contents of CR0, CR3, CR4, and the IA32\_SYSENTER\_CS, IA32\_SYSENTER\_ESP, and IA32\_SYSENTER\_EIP MSRs are saved into the corresponding fields. Bits 63:32 of the IA32\_SYSENTER\_CS MSR are not saved. On processors that do not support Intel 64 architecture, bits 63:32 of the IA32\_SYSENTER\_ESP and IA32\_SYSENTER\_EIP MSRs are not saved.
- If the “save debug controls” VM-exit control is 1, the contents of DR7 and the IA32\_DEBUGCTL MSR are saved into the corresponding fields. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always saved data into these fields.
- If the “save IA32\_PAT” VM-exit control is 1, the contents of the IA32\_PAT MSR are saved into the corresponding field.
- If the “save IA32\_EFER” VM-exit control is 1, the contents of the IA32\_EFER MSR are saved into the corresponding field.
- If the processor supports either the 1-setting of the “load IA32\_BNDCFGS” VM-entry control or that of the “clear IA32\_BNDCFGS” VM-exit control, the contents of the IA32\_BNDCFGS MSR are saved into the corresponding field.
- The value of the SMBASE field is undefined after all VM exits except SMM VM exits. See Section 34.15.2.

### 27.3.2 Saving Segment Registers and Descriptor-Table Registers

For each segment register (CS, SS, DS, ES, FS, GS, LDTR, or TR), the values saved for the base-address, segment-limit, and access rights are based on whether the register was unusable (see Section 24.4.1) before the VM exit:

- If the register was unusable, the values saved into the following fields are undefined: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in the access-rights field. The following exceptions apply:
  - CS.
    - The base-address and segment-limit fields are saved.
    - The L, D, and G bits are saved in the access-rights field.
  - SS.
    - DPL is saved in the access-rights field.
    - On processors that support Intel 64 architecture, bits 63:32 of the value saved for the base address are always zero.
  - DS and ES. On processors that support Intel 64 architecture, bits 63:32 of the values saved for the base addresses are always zero.
  - FS and GS. The base-address field is saved.
  - LDTR. The value saved for the base address is always canonical.
- If the register was not unusable, the values saved into the following fields are those which were in the register before the VM exit: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in access rights.

- Bits 31:17 and 11:8 in the access-rights field are always cleared. Bit 16 is set to 1 if and only if the segment is unusable.

The contents of the GDTR and IDTR registers are saved into the corresponding base-address and limit fields.

### 27.3.3 Saving RIP, RSP, and RFLAGS

The contents of the RIP, RSP, and RFLAGS registers are saved as follows:

- The value saved in the RIP field is determined by the nature and cause of the VM exit:
  - If the VM exit occurred in enclave mode, the value saved is the AEP of interrupted enclave thread (the remaining items do not apply).
  - If the VM exit occurs due to by an attempt to execute an instruction that causes VM exits unconditionally or that has been configured to cause a VM exit via the VM-execution controls, the value saved references that instruction.
  - If the VM exit is caused by an occurrence of an INIT signal, a start-up IPI (SIPI), or system-management interrupt (SMI), the value saved is that which was in RIP before the event occurred.
  - If the VM exit occurs due to the 1-setting of either the “interrupt-window exiting” VM-execution control or the “NMI-window exiting” VM-execution control, the value saved is that which would be in the register had the VM exit not occurred.
  - If the VM exit is due to an external interrupt, non-maskable interrupt (NMI), or hardware exception (as defined in Section 27.2.2), the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate,<sup>1</sup> or into the old task-state segment had the event been delivered through a task gate).
  - If the VM exit is due to a triple fault, the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate, or into the old task-state segment had the event been delivered through a task gate) had delivery of the double fault not encountered the nested exception that caused the triple fault.
  - If the VM exit is due to a software exception (due to an execution of INT3 or INTO) or a privileged software exception (due to an execution of INT1), the value saved references the INT3, INTO, or INT1 instruction that caused that exception.
  - Suppose that the VM exit is due to a task switch that was caused by execution of CALL, IRET, or JMP or by execution of a software interrupt (INT *n*), software exception (due to execution of INT3 or INTO), or privileged software exception (due to execution of INT1) that encountered a task gate in the IDT. The value saved references the instruction that caused the task switch (CALL, IRET, JMP, INT *n*, INT3, INTO, INT1).
  - Suppose that the VM exit is due to a task switch that was caused by a task gate in the IDT that was encountered for any reason except the direct access by a software interrupt or software exception. The value saved is that which would have been saved in the old task-state segment had the task switch completed normally.
  - If the VM exit is due to an execution of MOV to CR8 or WRMSR that reduced the value of bits 7:4 of VTPR (see Section 29.1.1) below that of TPR threshold VM-execution control field (see Section 29.1.2), the value saved references the instruction following the MOV to CR8 or WRMSR.
  - If the VM exit was caused by APIC-write emulation (see Section 29.4.3.2) that results from an APIC access as part of instruction execution, the value saved references the instruction following the one whose execution caused the APIC-write emulation.
- The contents of the RSP register are saved into the RSP field.
- With the exception of the resume flag (RF; bit 16), the contents of the RFLAGS register is saved into the RFLAGS field. RFLAGS.RF is saved as follows:
  - If the VM exit occurred in enclave mode, the value saved is 0 (the remaining items do not apply).

1. The reference here is to the full value of RIP before any truncation that would occur had the stack width been only 32 bits or 16 bits.



- If the VM exit is caused directly by an event that would normally be delivered through the IDT, the value saved is that which would appear in the saved RFLAGS image (either that which would be saved on the stack had the event been delivered through a trap or interrupt gate<sup>1</sup> or into the old task-state segment had the event been delivered through a task gate) had the event been delivered through the IDT. See below for VM exits due to task switches caused by task gates in the IDT.
- If the VM exit is caused by a triple fault, the value saved is that which the logical processor would have in RF in the RFLAGS register had the triple fault taken the logical processor to the shutdown state.
- If the VM exit is caused by a task switch (including one caused by a task gate in the IDT), the value saved is that which would have been saved in the RFLAGS image in the old task-state segment (TSS) had the task switch completed normally without exception.
- If the VM exit is caused by an attempt to execute an instruction that unconditionally causes VM exits or one that was configured to do with a VM-execution control, the value saved is 0.<sup>2</sup>
- For APIC-access VM exits and for VM exits caused by EPT violations, EPT misconfigurations, and page-modification log-full events, the value saved depends on whether the VM exit occurred during delivery of an event through the IDT:
  - If the VM exit stored 0 for bit 31 for IDT-vectoring information field (because the VM exit did not occur during delivery of an event through the IDT; see Section 27.2.3), the value saved is 1.
  - If the VM exit stored 1 for bit 31 for IDT-vectoring information field (because the VM exit did occur during delivery of an event through the IDT), the value saved is the value that would have appeared in the saved RFLAGS image had the event been delivered through the IDT (see above).
- For all other VM exits, the value saved is the value RFLAGS.RF had before the VM exit occurred.

### 27.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

- The activity-state field is saved with the logical processor's activity state before the VM exit.<sup>3</sup> See Section 27.1 for details of how events leading to a VM exit may affect the activity state.
- The interruptibility-state field is saved to reflect the logical processor's interruptibility before the VM exit.
  - See Section 27.1 for details of how events leading to a VM exit may affect this state.
  - VM exits that end outside system-management mode (SMM) save bit 2 (blocking by SMI) as 0 regardless of the state of such blocking before the VM exit.
  - Bit 3 (blocking by NMI) is treated specially if the "virtual NMIs" VM-execution control is 1. In this case, the value saved for this field does not indicate the blocking of NMIs but rather the state of virtual-NMI blocking.
  - Bit 4 (enclave interruption) is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.

- The pending debug exceptions field is saved as clear for all VM exits except the following:
  - A VM exit caused by an INIT signal, a machine-check exception, or a system-management interrupt (SMI).

1. The reference here is to the full value of RFLAGS before any truncation that would occur had the stack width been only 32 bits or 16 bits.
2. This is true even if RFLAGS.RF was 1 before the instruction was executed. If, in response to such a VM exit, a VM monitor re-enters the guest to re-execute the instruction that caused the VM exit (for example, after clearing the VM-execution control that caused the VM exit), the instruction may encounter a code breakpoint that has already been processed. A VM monitor can avoid this by setting the guest value of RFLAGS.RF to 1 before resuming guest software.
3. If this activity state was an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.

- A VM exit with basic exit reason “TPR below threshold”,<sup>1</sup> “virtualized EOI”, “APIC write”, or “monitor trap flag.”
- VM exits that are not caused by debug exceptions and that occur while there is MOV-SS blocking of debug exceptions.

For VM exits that do not clear the field, the value saved is determined as follows:

- Each of bits 3:0 may be set if it corresponds to a matched breakpoint. This may be true even if the corresponding breakpoint is not enabled in DR7.
- Suppose that a VM exit is due to an INIT signal, a machine-check exception, or an SMI; or that a VM exit has basic exit reason “TPR below threshold” or “monitor trap flag.” In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit.

If the VM exit occurs immediately after VM entry, the value saved may match that which was loaded on VM entry (see Section 26.6.3). Otherwise, the following items apply:

- Bit 12 (enabled breakpoint) is set to 1 in any of the following cases:
  - If there was at least one matched data or I/O breakpoint that was enabled in DR7.
  - If it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.6.3) and the VM exit occurred before those exceptions were either delivered or lost.
  - If the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). (This does not apply to VM exits with basic exit reason “monitor trap flag.”)

In other cases, bit 12 is cleared to 0.

- Bit 14 (BS) is set if RFLAGS.TF = 1 in either of the following cases:
  - IA32\_DEBUGCTL.BTF = 0 and the cause of a pending debug exception was the execution of a single instruction.
  - IA32\_DEBUGCTL.BTF = 1 and the cause of a pending debug exception was a taken branch.
- Bit 16 (RTM) is set if a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions had been enabled. (This does not apply to VM exits with basic exit reason “monitor trap flag.”)
- Suppose that a VM exit is due to another reason (but not a debug exception) and occurs while there is MOV-SS blocking of debug exceptions. In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit. If the VM exit occurs immediately after VM entry (no instructions were executed in VMX non-root operation), the value saved may match that which was loaded on VM entry (see Section 26.6.3). Otherwise, the following items apply:
  - Bit 12 (enabled breakpoint) is set to 1 if there was at least one matched data or I/O breakpoint that was enabled in DR7. Bit 12 is also set if it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.6.3) and the VM exit occurred before those exceptions were either delivered or lost. In other cases, bit 12 is cleared to 0.
  - The setting of bit 14 (BS) is implementation-specific. However, it is not set if RFLAGS.TF = 0 or IA32\_DEBUGCTL.BTF = 1.
- The reserved bits in the field are cleared.
- If the “save VMX-preemption timer value” VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 25.5.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value 0 if the timer expired during VM exit. (If the “save VMX-preemption timer value” VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)
- If the logical processor supports the 1-setting of the “enable EPT” VM-execution control, values are saved into the four (4) PDPTTE fields as follows:

---

1. This item includes VM exits that occur as a result of certain VM entries (Section 26.6.7).

- If the “enable EPT” VM-execution control is 1 and the logical processor was using PAE paging at the time of the VM exit, the PDPT values currently in use are saved:<sup>1</sup>
  - The values saved into bits 11:9 of each of the fields is undefined.
  - If the value saved into one of the fields has bit 0 (present) clear, the value saved into bits 63:1 of that field is undefined. That value need not correspond to the value that was loaded by VM entry or to any value that might have been loaded in VMX non-root operation.
  - If the value saved into one of the fields has bit 0 (present) set, the value saved into bits 63:12 of the field is a guest-physical address.
- If the “enable EPT” VM-execution control is 0 or the logical processor was not using PAE paging at the time of the VM exit, the values saved are undefined.

## 27.4 SAVING MSRS

After processor state is saved to the guest-state area, values of MSRs may be stored into the VM-exit MSR-store area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-store count) is processed in order by storing the value of the MSR indexed by bits 31:0 (as they would be read by RDMSR) into bits 127:64. Processing of an entry fails in either of the following cases:

- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be read only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32\_SMBASE is an MSR that can be read only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be saved on VM exits for model-specific reasons. A processor may prevent certain MSRs (based on the value of bits 31:0) from being stored on VM exits, even if they can normally be read by RDMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.
- Bits 63:32 of the entry are not all 0.
- An attempt to read the MSR indexed by bits 31:0 would cause a general-protection exception if executed via RDMSR with CPL = 0.

A VMX abort occurs if processing fails for any entry. See Section 27.7.

## 27.5 LOADING HOST STATE

Processor state is updated on VM exits in the following ways:

- Some state is loaded from or otherwise determined by the contents of the host-state area.
- Some state is determined by VM-exit controls.
- Some state is established in the same way on every VM exit.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order.

On processors that support Intel 64 architecture, the full values of each 64-bit field loaded (for example, the base address for GDTR) is loaded regardless of the mode of the logical processor before and after the VM exit.

The loading of host state is detailed in Section 27.5.1 to Section 27.5.5. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

---

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32\_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM exit functions as if the “enable EPT” VM-execution control were 0. See Section 24.6.2.

A logical processor is in IA-32e mode after a VM exit only if the “host address-space size” VM-exit control is 1. If the logical processor was in IA-32e mode before the VM exit and this control is 0, a VMX abort occurs. See Section 27.7.

In addition to loading host state, VM exits clear address-range monitoring (Section 27.5.6).

After the state loading described in this section, VM exits may load MSRs from the VM-exit MSR-load area (see Section 27.6). This loading occurs only after the state loading described in this section.

### 27.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
  - The following bits are not modified:
    - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 23.8).<sup>1</sup>
    - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width (they are cleared to 0).<sup>2</sup> (This item applies only to processors that support Intel 64 architecture.)
    - For CR4, any bits that are fixed in VMX operation (see Section 23.8).
  - CR4.PAE is set to 1 if the “host address-space size” VM-exit control is 1.
  - CR4.PCIDE is set to 0 if the “host address-space size” VM-exit control is 0.
- DR7 is set to 400H.
- The following MSRs are established as follows:
  - The IA32\_DEBUGCTL MSR is cleared to 00000000\_00000000H.
  - The IA32\_SYSENTER\_CS MSR is loaded from the IA32\_SYSENTER\_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  - IA32\_SYSENTER\_ESP MSR and IA32\_SYSENTER\_EIP MSR are loaded from the IA32\_SYSENTER\_ESP field and the IA32\_SYSENTER\_EIP field, respectively.
 

If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

If the processor does support the Intel 64 architecture and the processor supports  $N < 64$  linear-address bits, each of bits 63:N is set to the value of bit  $N-1$ .<sup>3</sup>
  - The following steps are performed on processors that support Intel 64 architecture:
    - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 27.5.2).
    - The LMA and LME bits in the IA32\_EFER MSR are each loaded with the setting of the “host address-space size” VM-exit control.
  - If the “load IA32\_PERF\_GLOBAL\_CTRL” VM-exit control is 1, the IA32\_PERF\_GLOBAL\_CTRL MSR is loaded from the IA32\_PERF\_GLOBAL\_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the “load IA32\_PAT” VM-exit control is 1, the IA32\_PAT MSR is loaded from the IA32\_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.

1. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.

2. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

3. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- If the “load IA32\_EFER” VM-exit control is 1, the IA32\_EFER MSR is loaded from the IA32\_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.
- If the “clear IA32\_BNDCFGS” VM-exit control is 1, the IA32\_BNDCFGS MSR is cleared to 00000000\_00000000H; otherwise, it is not modified.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 27.6.

## 27.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified Section 26.3.1.2 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).
- The base address is set as follows:
  - CS. Cleared to zero.
  - SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.
  - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.
 

If the processor supports the Intel 64 architecture and the processor supports  $N < 64$  linear-address bits, each of bits 63:N is set to the value of bit N-1.<sup>1</sup> The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
  - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports  $N < 64$  linear-address bits, each of bits 63:N is set to the value of bit N-1.
- The segment limit is set as follows:
  - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFFH and a G-bit setting of 1).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
  - TR. Set to 00000067H.
- The type field and S bit are set as follows:
  - CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
  - TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).
- The DPL is set as follows:
  - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.
  - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.
- The P bit is set as follows:
  - CS, TR. Set to 1.
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the “host address-space size” VM-exit control. Because the value of this control is also loaded into IA32\_EFER.LMA (see Section 27.5.1), no VM exit is ever to compatibility mode (which requires IA32\_EFER.LMA = 1 and CS.L = 0).
- D/B.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- CS. Loaded with the inverse of the setting of the “host address-space size” VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
- SS. Set to 1.
- DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- TR. Set to 0.
- G.
  - CS. Set to 1.
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
  - TR. Set to 0.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined (although the base address is always canonical).

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports  $N < 64$  linear-address bits, each of bits 63:N of each base address is set to the value of bit N–1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

### 27.5.3 Loading Host RIP, RSP, and RFLAGS

RIP and RSP are loaded from the RIP field and the RSP field, respectively. RFLAGS is cleared, except bit 1, which is always set.

### 27.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries

If  $CR0.PG = 1$ ,  $CR4.PAE = 1$ , and  $IA32\_EFER.LMA = 0$ , the logical processor uses **PAE paging**. See Section 4.4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.<sup>1</sup> When in PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTes and, if they are valid, loads them into the processor (into internal, non-architectural registers).

A VM exit is to a VMM that uses PAE paging if (1) bit 5 (corresponding to CR4.PAE) is set in the CR4 field in the host-state area of the VMCS; and (2) the “host address-space size” VM-exit control is 0. Such a VM exit may check the validity of the PDPTes referenced by the CR3 field in the host-state area of the VMCS. Such a VM exit must check their validity if either (1) PAE paging was not in use before the VM exit; or (2) the value of CR3 is changing as a result of the VM exit. A VM exit to a VMM that does not use PAE paging must not check the validity of the PDPTes.

A VM exit that checks the validity of the PDPTes uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the PDPTes that would be loaded (e.g., because a reserved bit is set), a VMX abort occurs (see Section 27.7). If a VM exit to a VMM that uses PAE does not cause a VMX abort, the PDPTes are loaded into the processor as would MOV to CR3, using the value of CR3 being load by the VM exit.

### 27.5.5 Updating Non-Register State

VM exits affect the non-register state of a logical processor as follows:

- A logical processor is always in the active state after a VM exit.
- Event blocking is affected as follows:
  - There is no blocking by STI or by MOV SS after a VM exit.

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- VM exits caused directly by non-maskable interrupts (NMIs) cause blocking by NMI (see Table 24-3). Other VM exits do not affect blocking by NMI. (See Section 27.1 for the case in which an NMI causes a VM exit indirectly.)
- There are no pending debug exceptions after a VM exit.

Section 28.3 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM exits invalidate cached mappings:

- If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP).
- VM exits are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

## 27.5.6 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 8.10.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. VM exits clear any address-range monitoring that may be in effect.

## 27.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32\_FS\_BASE MSR) or C0000101H (the IA32\_GS\_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32\_SMM\_MONITOR\_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.
- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.<sup>1</sup>

If processing fails for any entry, a VMX abort occurs. See Section 27.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

## 27.7 VMX ABORTS

A problem encountered during a VM exit leads to a VMX abort. A VMX abort takes a logical processor into a shut-down state as described below.

- 
1. Note the following about processors that support Intel 64 architecture. If CR0.PG = 1, WRMSR to the IA32\_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CR0.PG is always 1 in VMX operation, the IA32\_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.

A VMX abort does not modify the VMCS data in the VMCS region of any active VMCS. The contents of these data are thus suspect after the VMX abort.

On a VMX abort, a logical processor saves a nonzero 32-bit VMX-abort indicator field at byte offset 4 in the VMCS region of the VMCS whose misconfiguration caused the failure (see Section 24.2). The following values are used:

1. There was a failure in saving guest MSRs (see Section 27.4).
2. Host checking of the page-directory-pointer-table entries (PDPTes) failed (see Section 27.5.4).
3. The current VMCS has been corrupted (through writes to the corresponding VMCS region) in such a way that the logical processor cannot complete the VM exit properly.
4. There was a failure on loading host MSRs (see Section 27.6).
5. There was a machine-check event during VM exit (see Section 27.8).
6. The logical processor was in IA-32e mode before the VM exit and the “host address-space size” VM-entry control was 0 (see Section 27.5).

Some of these causes correspond to failures during the loading of state from the host-state area. Because the loading of such state may be done in any order (see Section 27.5) a VM exit that might lead to a VMX abort for multiple reasons (for example, the current VMCS may be corrupt and the host PDPTes might not be properly configured). In such cases, the VMX-abort indicator could correspond to any one of those reasons.

A logical processor never reads the VMX-abort indicator in a VMCS region and writes it only with one of the non-zero values mentioned above. The VMX-abort indicator allows software on one logical processor to diagnose the VMX-abort on another. For this reason, it is recommended that software running in VMX root operation zero the VMX-abort indicator in the VMCS region of any VMCS that it uses.

After saving the VMX-abort indicator, operation of a logical processor experiencing a VMX abort depends on whether the logical processor is in SMX operation:<sup>1</sup>

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000DH, indicating “VMX abort.” See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*.
- If the logical processor is outside SMX operation, it issues a special bus cycle (to notify the chipset) and enters the **VMX-abort shutdown state**. RESET is the only event that wakes a logical processor from the VMX-abort shutdown state. The following events do not affect a logical processor in this state: machine-check events; INIT signals; external interrupts; non-maskable interrupts (NMIs); start-up IPIs (SIPIs); and system-management interrupts (SMIs).

## 27.8 MACHINE-CHECK EVENTS DURING VM EXIT

If a machine-check event occurs during VM exit, one of the following occurs:

- The machine-check event is handled as if it occurred before the VM exit:
  - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:<sup>2</sup>
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating “unrecoverable machine-check condition.”
    - If the logical processor is outside SMX operation, it goes to the shutdown state.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

2. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.



- If CR4.MCE = 1, a machine-check exception (#MC) is generated:
  - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
  - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- The machine-check event is handled after VM exit completes:
  - If the VM exit ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    - If the logical processor is in SMX operation, an Intel<sup>®</sup> TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If the VM exit ends with CR4.MCE = 1, a machine-check exception (#MC) is delivered through the host IDT.
- A VMX abort is generated (see Section 27.7). The logical processor blocks events as done normally in VMX abort. The VMX abort indicator is 5, for “machine-check event during VM exit.”

The first option is not used if the machine-check event occurs after any host state has been loaded. The second option is used only if VM entry is able to load all host state.



## 17. Updates to Chapter 34, Volume 3C

Change bars show changes to Chapter 34 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----

Changes to this chapter: Minor update to Section 34.6 "Exceptions and Interrupts Within SMM".

This chapter describes aspects of IA-64 and IA-32 architecture used in system management mode (SMM).

SMM provides an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code. It was introduced into the IA-32 architecture in the Intel386 SL processor (a mobile specialized version of the Intel386 processor). It is also available in the Pentium M, Pentium 4, Intel Xeon, P6 family, and Pentium and Intel486 processors (beginning with the enhanced versions of the Intel486 SL and Intel486 processors).

### 34.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

When SMM is invoked through a system management interrupt (SMI), the processor saves the current state of the processor (the processor's context), then switches to a separate operating environment defined by a new address space. The system management software executive (SMI handler) starts execution in that environment, and the critical code and data of the SMI handler reside in a physical memory region (SMRAM) within that address space. While in SMM, the processor executes SMI handler code to perform operations such as powering down unused disk drives or monitors, executing proprietary code, or placing the whole system in a suspended state. When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.
- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- The RSM instruction can be executed only in SMM.

Section 34.3 describes transitions into and out of SMM. The execution environment after entering SMM is in real-address mode with paging disabled ( $CR0.PE = CR0.PG = 0$ ). In this initial execution environment, the SMI handler can address up to 4 GBytes of memory and can execute all I/O and system instructions. Section 34.5 describes in detail the initial SMM execution environment for an SMI handler and operation within that environment. The SMI handler may subsequently switch to other operating modes while remaining in SMM.

#### NOTES

Software developers should be aware that, even if a logical processor was using the physical-address extension (PAE) mechanism (introduced in the P6 family processors) or was in IA-32e mode before an SMI, this will not be the case after the SMI is delivered. This is because delivery of an SMI disables paging (see Table 34-4). (This does not apply if the dual-monitor treatment of SMIs and SMM is active; see Section 34.15.)

#### 34.1.1 System Management Mode and VMX Operation

Traditionally, SMM services system management interrupts and then resumes program execution (back to the software stack consisting of executive and application software; see Section 34.2 through Section 34.13).

A virtual machine monitor (VMM) using VMX can act as a host to multiple virtual machines and each virtual machine can support its own software stack of executive and application software. On processors that support VMX, virtual-machine extensions may use system-management interrupts (SMIs) and system-management mode (SMM) in one of two ways:

- **Default treatment.** System firmware handles SMIs. The processor saves architectural states and critical states relevant to VMX operation upon entering SMM. When the firmware completes servicing SMIs, it uses RSM to resume VMX operation.
- **Dual-monitor treatment.** Two VM monitors collaborate to control the servicing of SMIs: one VMM operates outside of SMM to provide basic virtualization in support for guests; the other VMM operates inside SMM (while in VMX operation) to support system-management functions. The former is referred to as **executive monitor**, the latter **SMM-transfer monitor (STM)**.<sup>1</sup>

The default treatment is described in Section 34.14, “Default Treatment of SMIs and SMM with VMX Operation and SMX Operation”. Dual-monitor treatment of SMM is described in Section 34.15, “Dual-Monitor Treatment of SMIs and SMM”.

## 34.2 SYSTEM MANAGEMENT INTERRUPT (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor’s interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

### NOTES

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI.

An SMI may be blocked for one instruction following execution of STI, MOV to SS, or POP into SS.

## 34.3 SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES

Figure 2-3 shows how the processor moves between SMM and the other processor operating modes (protected, real-address, and virtual-8086). Signaling an SMI while the processor is in real-address, protected, or virtual-8086 modes always causes the processor to switch to SMM. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

### 34.3.1 Entering SMM

The processor always handles an SMI on an architecturally defined “interruptible” point in program execution (which is commonly at an IA-32 architecture instruction boundary). When the processor receives an SMI, it waits for all instructions to retire and for all stores to complete. The processor then saves its current context in SMRAM (see Section 34.4), enters SMM, and begins to execute the SMI handler.

Upon entering SMM, the processor signals external hardware that SMI handling has begun. The signaling mechanism used is implementation dependent. For the P6 family processors, an SMI acknowledge transaction is gener-

---

1. The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

ated on the system bus and the multiplexed status signal EXF4 is asserted each time a bus transaction is generated while the processor is in SMM. For the Pentium and Intel486 processors, the SMIACK# pin is asserted.

An SMI has a greater priority than debug exceptions and external interrupts. Thus, if an NMI, maskable hardware interrupt, or a debug exception occurs at an instruction boundary along with an SMI, only the SMI is handled. Subsequent SMI requests are not acknowledged while the processor is in SMM. The first SMI interrupt request that occurs while the processor is in SMM (that is, after SMM has been acknowledged to external hardware) is latched and serviced when the processor exits SMM with the RSM instruction. The processor will latch only one SMI while in SMM.

See Section 34.5 for a detailed description of the execution environment when in SMM.

### 34.3.2 Exiting From SMM

The only way to exit SMM is to execute the RSM instruction. The RSM instruction is only available to the SMI handler; if the processor is not in SMM, attempts to execute the RSM instruction result in an invalid-opcode exception (#UD) being generated.

The RSM instruction restores the processor's context by loading the state save image from SMRAM back into the processor's registers. The processor then returns an SMIACK transaction on the system bus and returns program control back to the interrupted program.

Upon successful completion of the RSM instruction, the processor signals external hardware that SMM has been exited. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is no longer generated on bus cycles. For the Pentium and Intel486 processors, the SMIACK# pin is deserted.

If the processor detects invalid state information saved in the SMRAM, it enters the shutdown state and generates a special bus cycle to indicate it has entered shutdown state. Shutdown happens only in the following situations:

- A reserved bit in control register CR4 is set to 1 on a write to CR4. This error should not happen unless SMI handler code modifies reserved areas of the SMRAM saved state map (see Section 34.4.1). CR4 is saved in the state map in a reserved location and cannot be read or modified in its saved state.
- An illegal combination of bits is written to control register CR0, in particular PG set to 1 and PE set to 0, or NW set to 1 and CD set to 0.
- CR4.PCIDE would be set to 1 and IA32\_EFER.LMA to 0.
- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMI handler to uninhibit them (see Section 34.8).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 34.10). Also, the SMBASE address can be changed on a return from SMM (see Section 34.11).

## 34.4 SMRAM

Upon entering SMM, the processor switches to a new address space. Because paging is disabled upon entering SMM, this initial address space maps all memory accesses to the low 4 GBytes of the processor's physical address space. The SMI handler's critical code and data reside in a memory region referred to as system-management RAM (SMRAM). The processor uses a pre-defined region within SMRAM to save the processor's pre-SMI context. SMRAM can also be used to store system management information (such as the system configuration and specific information about powered-down devices) and OEM-specific information.

The default SMRAM size is 64 KBytes beginning at a base physical address in physical memory called the SMBASE (see Figure 34-1). The SMBASE default value following a hardware reset is 30000H. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 8000H]. It stores the processor's state in the area from [SMBASE + FE00H] to [SMBASE + FFFFH]. See Section 34.4.1 for a description of the mapping of the state save area.

The system logic is minimally required to decode the physical address range for the SMRAM from [SMBASE + 8000H] to [SMBASE + FFFFH]. A larger area can be decoded if needed. The size of this SMRAM can be between 32 KBytes and 4 GBytes.

The location of the SMRAM can be changed by changing the SMBASE value (see Section 34.11). It should be noted that all processors in a multiple-processor system are initialized with the same SMBASE value (30000H). Initialization software must sequentially place each processor in SMM and change its SMBASE so that it does not overlap those of other processors.

The actual physical location of the SMRAM can be in system memory or in a separate RAM memory. The processor generates an SMI acknowledge transaction (P6 family processors) or asserts the SMIACK# pin (Pentium and Intel486 processors) when the processor receives an SMI (see Section 34.3.1).

System logic can use the SMI acknowledge transaction or the assertion of the SMIACK# pin to decode accesses to the SMRAM and redirect them (if desired) to specific SMRAM memory. If a separate RAM memory is used for SMRAM, system logic should provide a programmable method of mapping the SMRAM into system memory space when the processor is not in SMM. This mechanism will enable start-up procedures to initialize the SMRAM space (that is, load the SMI handler) before executing the SMI handler during SMM.

### 34.4.1 SMRAM State Save Map

When an IA-32 processor that does not support Intel 64 architecture initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7E00H]. Table 34-1 shows the state save map. The offset in column 1 is relative to the SMBASE value plus 8000H. Reserved spaces should not be used by software.

Some of the registers in the SMRAM state save area (marked YES in column 3) may be read and changed by the SMI handler, with the changed values restored to the processor registers by the RSM instruction. Some register images are read-only, and must not be modified (modifying these registers will result in unpredictable behavior). An SMI handler should not rely on any values stored in an area that is marked as reserved.

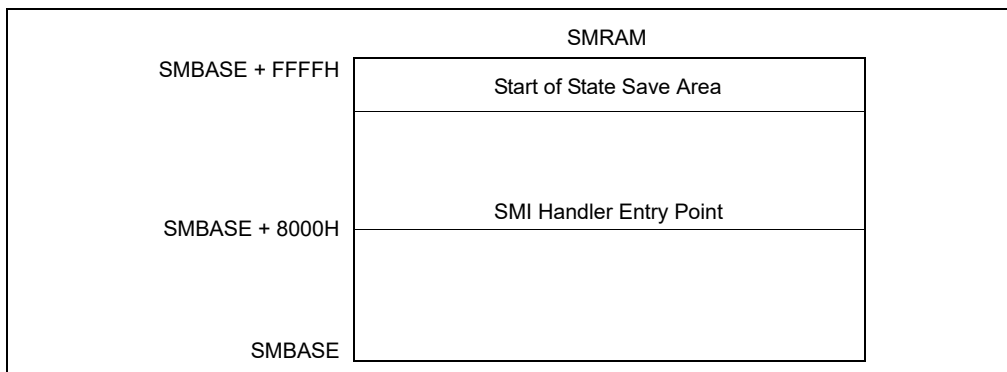


Figure 34-1. SMRAM Usage

Table 34-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FFCH	CR0	No
7FF8H	CR3	No
7FF4H	EFLAGS	Yes
7FF0H	EIP	Yes
7FECH	EDI	Yes
7FE8H	ESI	Yes
7FE4H	EBP	Yes
7FE0H	ESP	Yes
7FDCH	EBX	Yes
7FD8H	EDX	Yes
7FD4H	ECX	Yes
7FD0H	EAX	Yes
7FCCH	DR6	No
7FC8H	DR7	No
7FC4H	TR <sup>1</sup>	No
7FC0H	Reserved	No
7FBCH	GS <sup>1</sup>	No
7FB8H	FS <sup>1</sup>	No
7FB4H	DS <sup>1</sup>	No
7FB0H	SS <sup>1</sup>	No
7FACH	CS <sup>1</sup>	No
7FA8H	ES <sup>1</sup>	No
7FA4H	I/O State Field, see Section 34.7	No
7FA0H	I/O Memory Address Field, see Section 34.7	No
7F9FH-7F03H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes
7EF7H - 7E00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

The following registers are saved (but not readable) and restored upon exiting SMM:

- Control register CR4. (This register is cleared to all 0s when entering SMM).
- The hidden segment descriptor information stored in segment registers CS, DS, ES, FS, GS, and SS.

If an SMI request is issued for the purpose of powering down the processor, the values of all reserved locations in the SMM state save must be saved to nonvolatile memory.

The following state is not automatically saved and restored following an SMI and the RSM instruction, respectively:



- Debug registers DR0 through DR3.
- The x87 FPU registers.
- The MTRRs.
- Control register CR2.
- The model-specific registers (for the P6 family and Pentium processors) or test registers TR3 through TR7 (for the Pentium and Intel486 processors).
- The state of the trap controller.
- The machine-check architecture registers.
- The APIC internal interrupt state (ISR, IRR, etc.).
- The microcode update state.

If an SMI is used to power down the processor, a power-on reset will be required before returning to SMM, which will reset much of this state back to its default values. So an SMI handler that is going to trigger power down should first read these registers listed above directly, and save them (along with the rest of RAM) to nonvolatile storage. After the power-on reset, the continuation of the SMI handler should restore these values, along with the rest of the system's state. Anytime the SMI handler changes these registers in the processor, it must also save and restore them.

### NOTES

A small subset of the MSRs (such as, the time-stamp counter and performance-monitoring counters) are not arbitrarily writable and therefore cannot be saved and restored. SMM-based power-down and restoration should only be performed with operating systems that do not use or rely on the values of these registers.

Operating system developers should be aware of this fact and insure that their operating-system assisted power-down and restoration software is immune to unexpected changes in these register values.

#### 34.4.1.1 SMRAM State Save Map and Intel 64 Architecture

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H].

Support for Intel 64 architecture is reported by CPUID.80000001:EDX[29] = 1. The layout of the SMRAM state save map is shown in Table 34-3.

Additionally, the SMRAM state save map shown in Table 34-3 also applies to processors with the following CPUID signatures listed in Table 34-2, irrespective of the value in CPUID.80000001:EDX[29].

**Table 34-2. Processor Signatures and 64-bit SMRAM State Save Map Format**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9xxx, Intel Core 2 Duo processors E8000, T9000,
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors
06_1CH	45 nm Intel® Atom™ processors

Table 34-3. SMRAM State Save Map for Intel 64 Architecture

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FF8H	CR0	No
7FF0H	CR3	No
7FE8H	RFLAGS	Yes
7FE0H	IA32_EFER	Yes
7FD8H	RIP	Yes
7FD0H	DR6	No
7FC8H	DR7	No
7FC4H	TR SEL <sup>1</sup>	No
7FC0H	LDTR SEL <sup>1</sup>	No
7FBCH	GS SEL <sup>1</sup>	No
7FB8H	FS SEL <sup>1</sup>	No
7FB4H	DS SEL <sup>1</sup>	No
7FB0H	SS SEL <sup>1</sup>	No
7FACH	CS SEL <sup>1</sup>	No
7FA8H	ES SEL <sup>1</sup>	No
7FA4H	IO_MISC	No
7F9CH	IO_MEM_ADDR	No
7F94H	RDI	Yes
7F8CH	RSI	Yes
7F84H	RBP	Yes
7F7CH	RSP	Yes
7F74H	RBX	Yes
7F6CH	RDX	Yes
7F64H	RCX	Yes
7F5CH	RAX	Yes
7F54H	R8	Yes
7F4CH	R9	Yes
7F44H	R10	Yes
7F3CH	R11	Yes
7F34H	R12	Yes
7F2CH	R13	Yes
7F24H	R14	Yes
7F1CH	R15	Yes
7F1BH-7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes

Table 34-3. SMRAM State Save Map for Intel 64 Architecture (Contd.)

Offset (Added to SMBASE + 8000H)	Register	Writable?
7EF7H - 7EE4H	Reserved	No
7EE0H	Setting of "enable EPT" VM-execution control	No
7ED8H	Value of EPTP VM-execution control field	No
7ED7H - 7EA0H	Reserved	No
7E9CH	LDT Base (lower 32 bits)	No
7E98H	Reserved	No
7E94H	IDT Base (lower 32 bits)	No
7E90H	Reserved	No
7E8CH	GDT Base (lower 32 bits)	No
7E8BH - 7E44H	Reserved	No
7E40H	CR4	No
7E3FH - 7DF0H	Reserved	No
7DE8H	IO_RIP	Yes
7DE7H - 7DDCH	Reserved	No
7DD8H	IDT Base (Upper 32 bits)	No
7DD4H	LDT Base (Upper 32 bits)	No
7DD0H	GDT Base (Upper 32 bits)	No
7DCFH - 7C00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

### 34.4.2 SMRAM Caching

An IA-32 processor does not automatically write back and invalidate its caches before entering SMM or before exiting SMM. Because of this behavior, care must be taken in the placement of the SMRAM in system memory and in the caching of the SMRAM to prevent cache incoherence when switching back and forth between SMM and protected mode operation. Any of the following three methods of locating the SMRAM in system memory will guarantee cache coherency.

- Place the SMRAM in a dedicated section of system memory that the operating system and applications are prevented from accessing. Here, the SMRAM can be designated as cacheable (WB, WT, or WC) for optimum processor performance, without risking cache incoherence when entering or exiting SMM.
- Place the SMRAM in a section of memory that overlaps an area used by the operating system (such as the video memory), but designate the SMRAM as uncacheable (UC). This method prevents cache access when in SMM to maintain cache coherency, but the use of uncacheable memory reduces the performance of SMM code.
- Place the SMRAM in a section of system memory that overlaps an area used by the operating system and/or application code, but explicitly flush (write back and invalidate) the caches upon entering and exiting SMM mode. This method maintains cache coherency, but incurs the overhead of two complete cache flushes.

For Pentium 4, Intel Xeon, and P6 family processors, a combination of the first two methods of locating the SMRAM is recommended. Here the SMRAM is split between an overlapping and a dedicated region of memory. Upon entering SMM, the SMRAM space that is accessed overlaps video memory (typically located in low memory). This SMRAM section is designated as UC memory. The initial SMM code then jumps to a second SMRAM section that is located in a dedicated region of system memory (typically in high memory). This SMRAM section can be cached for optimum processor performance.

For systems that explicitly flush the caches upon entering SMM (the third method described above), the cache flush can be accomplished by asserting the FLUSH# pin at the same time as the request to enter SMM (generally initiated by asserting the SMI# pin). The priorities of the FLUSH# and SMI# pins are such that the FLUSH# is serviced first. To guarantee this behavior, the processor requires that the following constraints on the interaction of FLUSH# and SMI# be met. In a system where the FLUSH# and SMI# pins are synchronous and the set up and hold times are met, then the FLUSH# and SMI# pins may be asserted in the same clock. In asynchronous systems, the FLUSH# pin must be asserted at least one clock before the SMI# pin to guarantee that the FLUSH# pin is serviced first.

Upon leaving SMM (for systems that explicitly flush the caches), the WBINVD instruction should be executed prior to leaving SMM to flush the caches.

## NOTES

In systems based on the Pentium processor that use the FLUSH# pin to write back and invalidate cache contents before entering SMM, the processor will prefetch at least one cache line in between when the Flush Acknowledge cycle is run and the subsequent recognition of SMI# and the assertion of SMIACK#.

It is the obligation of the system to ensure that these lines are not cached by returning KEN# inactive to the Pentium processor.

### 34.4.2.1 System Management Range Registers (SMRR)

SMI handler code and data stored by SMM code resides in SMRAM. The SMRR interface is an enhancement in Intel 64 architecture to limit cacheable reference of addresses in SMRAM to code running in SMM. The SMRR interface can be configured only by code running in SMM. Details of SMRR is described in Section 11.11.2.4.

## 34.5 SMI HANDLER EXECUTION ENVIRONMENT

Section 34.5.1 describes the initial execution environment for an SMI handler. An SMI handler may re-configure its execution environment to other supported operating modes. Section 34.5.2 discusses modifications an SMI handler can make to its execution environment.

### 34.5.1 Initial SMM Execution Environment

After saving the current context of the processor, the processor initializes its core registers to the values shown in Table 34-4. Upon entering SMM, the PE and PG flags in control register CR0 are cleared, which places the processor in an environment similar to real-address mode. The differences between the SMM execution environment and the real-address mode execution environment are as follows:

- The addressable address space ranges from 0 to FFFFFFFFH (4 GBytes).
- The normal 64-KByte segment limit for real-address mode is increased to 4 GBytes.
- The default operand and address sizes are set to 16 bits, which restricts the addressable SMRAM address space to the 1-MByte real-address mode limit for native real-address-mode code. However, operand-size and address-size override prefixes can be used to access the address space beyond the 1-MByte.

**Table 34-4. Processor Register Initialization in SMM**

Register	Contents
General-purpose registers	Undefined
EFLAGS	00000002H
EIP	00008000H
CS selector	SMM Base shifted right 4 bits (default 3000H)
CS base	SMM Base (default 30000H)
DS, ES, FS, GS, SS Selectors	0000H

**Table 34-4. Processor Register Initialization in SMM**

DS, ES, FS, GS, SS Bases	000000000H
DS, ES, FS, GS, SS Limits	0FFFFFFFH
CR0	PE, EM, TS, and PG flags set to 0; others unmodified
CR4	Cleared to zero
DR6	Undefined
DR7	00000400H

- Near jumps and calls can be made to anywhere in the 4-GByte address space if a 32-bit operand-size override prefix is used. Due to the real-address-mode style of base-address formation, a far call or jump cannot transfer control to a segment with a base address of more than 20 bits (1 MByte). However, since the segment limit in SMM is 4 GBytes, offsets into a segment that go beyond the 1-MByte limit are allowed when using 32-bit operand-size override prefixes. Any program control transfer that does not have a 32-bit operand-size override prefix truncates the EIP value to the 16 low-order bits.
- Data and the stack can be located anywhere in the 4-GByte address space, but can be accessed only with a 32-bit address-size override if they are located above 1 MByte. As with the code segment, the base address for a data or stack segment cannot be more than 20 bits.

The value in segment register CS is automatically set to the default of 30000H for the SMBASE shifted 4 bits to the right; that is, 3000H. The EIP register is set to 8000H. When the EIP value is added to shifted CS value (the SMBASE), the resulting linear address points to the first instruction of the SMI handler.

The other segment registers (DS, SS, ES, FS, and GS) are cleared to 0 and their segment limits are set to 4 GBytes. In this state, the SMRAM address space may be treated as a single flat 4-GByte linear address space. If a segment register is loaded with a 16-bit value, that value is then shifted left by 4 bits and loaded into the segment base (hidden part of the segment register). The limits and attributes are not modified.

Maskable hardware interrupts, exceptions, NMI interrupts, SMI interrupts, A20M interrupts, single-step traps, breakpoint traps, and INIT operations are inhibited when the processor enters SMM. Maskable hardware interrupts, exceptions, single-step traps, and breakpoint traps can be enabled in SMM if the SMM execution environment provides and initializes an interrupt table and the necessary interrupt and exception handlers (see Section 34.6).

### 34.5.2 SMI Handler Operating Mode Switching

Within SMM, an SMI handler may change the processor's operating mode (e.g., to enable PAE paging, enter 64-bit mode, etc.) after it has made proper preparation and initialization to do so. For example, if switching to 32-bit protected mode, the SMI handler should follow the guidelines provided in Chapter 9, "Processor Management and Initialization". If the SMI handler does wish to change operating mode, it is responsible for executing the appropriate mode-transition code after each SMI.

It is recommended that the SMI handler make use of all means available to protect the integrity of its critical code and data. In particular, it should use the system-management range register (SMRR) interface if it is available (see Section 11.11.2.4). The SMRR interface can protect only the first 4 GBytes of the physical address space. The SMI handler should take that fact into account if it uses operating modes that allow access to physical addresses beyond that 4-GByte limit (e.g. PAE paging or 64-bit mode).

Execution of the RSM instruction restores the pre-SMI processor state from the SMRAM state-state map (see Section 34.4.1) into which it was stored when the processor entered SMM. (The SMBASE field in the SMRAM state-state map does not determine the state following RSM but rather the initial environment following the next entry to SMM.) Any required change to operating mode is performed by the RSM instruction; there is no need for the SMI handler to change modes explicitly prior to executing RSM.

## 34.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM

When the processor enters SMM, all hardware interrupts are disabled in the following manner:

- The IF flag in the EFLAGS register is cleared, which inhibits maskable hardware interrupts from being generated.
- The TF flag in the EFLAGS register is cleared, which disables single-step traps.
- Debug register DR7 is cleared, which disables breakpoint traps. (This action prevents a debugger from accidentally breaking into an SMI handler if a debug breakpoint is set in normal address space that overlays code or data in SMRAM.)
- NMI, SMI, and A20M interrupts are blocked by internal SMM logic. (See Section 34.8 for more information about how NMIs are handled in SMM.)

Software-invoked interrupts and exceptions can still occur, and maskable hardware interrupts can be enabled by setting the IF flag. Intel recommends that SMM code be written in so that it does not invoke software interrupts (with the INT *n*, INTO, INT1, INT3, or BOUND instructions) or generate exceptions.

If the SMI handler requires interrupt and exception handling, an SMM interrupt table and the necessary exception and interrupt handlers must be created and initialized from within SMM. Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and software interrupts will result in unpredictable processor behavior.

The following restrictions apply when designing SMM interrupt and exception-handling facilities:

- The interrupt table should be located at linear address 0 and must contain real-address mode style interrupt vectors (4 bytes containing CS and IP).
- Due to the real-address mode style of base address formation, an interrupt or exception cannot transfer control to a segment with a base address of more than 20 bits.
- An interrupt or exception cannot transfer control to a segment offset of more than 16 bits (64 KBytes).
- When an exception or interrupt occurs, only the 16 least-significant bits of the return address (EIP) are pushed onto the stack. If the offset of the interrupted procedure is greater than 64 KBytes, it is not possible for the interrupt/exception handler to return control to that procedure. (One solution to this problem is for a handler to adjust the return address on the stack.)
- The SMBASE relocation feature affects the way the processor will return from an interrupt or exception generated while the SMI handler is executing. For example, if the SMBASE is relocated to above 1 MByte, but the exception handlers are below 1 MByte, a normal return to the SMI handler is not possible. One solution is to provide the exception handler with a mechanism for calculating a return address above 1 MByte from the 16-bit return address on the stack, then use a 32-bit far call to return to the interrupted procedure.
- If an SMI handler needs access to the debug trap facilities, it must insure that an SMM accessible debug handler is available and save the current contents of debug registers DR0 through DR3 (for later restoration). Debug registers DR0 through DR3 and DR7 must then be initialized with the appropriate values.
- If an SMI handler needs access to the single-step mechanism, it must insure that an SMM accessible single-step handler is available, and then set the TF flag in the EFLAGS register.
- If the SMI design requires the processor to respond to maskable hardware interrupts or software-generated interrupts while in SMM, it must ensure that SMM accessible interrupt handlers are available and then set the IF flag in the EFLAGS register (using the STI instruction). Software interrupts are not blocked upon entry to SMM, so they do not need to be enabled.

## 34.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

When coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

### 34.7.1 I/O State Implementation

Within the extended SMM state save map, a bit (IO\_SMI) is provided that is set only when an SMI is either taken immediately after a *successful* I/O instruction or is taken after a *successful* iteration of a REP I/O instruction (the *successful* notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO\_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO\_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

The IO\_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO\_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 34-5). The IO\_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO\_SMI bit is not set.

**Table 34-5. I/O Instruction Information in the SMM State Save Map**

State (SMM Rev. ID: 30004H or higher)	Format								
	31	16	15	8	7	4	3	1	0
I/O State Field SMRAM offset 7FA4		I/O Port		Reserved		I/O Type		I/O Length	IO_SMI
	31								0
I/O Memory Address Field SMRAM offset 7FA0	I/O Memory Address								

When IO\_SMI is set, the other fields may be interpreted as follows:

- I/O length:
  - 001 – Byte
  - 010 – Word
  - 100 – Dword
- I/O instruction type (Table 34-6)

**Table 34-6. I/O Instruction Type Encodings**

Instruction	Encoding
IN Immediate	1001
IN DX	0001
OUT Immediate	1000
OUT DX	0000
INS	0011
OUTS	0010
REP INS	0111
REP OUTS	0110

## 34.8 NMI HANDLING WHILE IN SMM

NMI interrupts are blocked upon entry to the SMI handler. If an NMI request occurs during the SMI handler, it is latched and serviced after the processor exits SMM. Only one NMI request will be latched during the SMI handler. If an NMI request is pending when the processor executes the RSM instruction, the NMI is serviced before the next instruction of the interrupted code sequence. This assumes that NMIs were not blocked before the SMI occurred. If NMIs were blocked before the SMI occurred, they are blocked after execution of RSM.

Although NMI requests are blocked when the processor enters SMM, they may be enabled through software by executing an IRET instruction. If the SMI handler requires the use of NMI interrupts, it should invoke a dummy interrupt service routine for the purpose of executing an IRET instruction. Once an IRET instruction is executed, NMI interrupt requests are serviced in the same “real mode” manner in which they are handled outside of SMM.

A special case can occur if an SMI handler nests inside an NMI handler and then another NMI occurs. During NMI interrupt handling, NMI interrupts are disabled, so normally NMI interrupts are serviced and completed with an IRET instruction one at a time. When the processor enters SMM while executing an NMI handler, the processor saves the SMRAM state save map but does not save the attribute to keep NMI interrupts disabled. Potentially, an NMI could be latched (while in SMM or upon exit) and serviced upon exit of SMM even though the previous NMI handler has still not completed. One or more NMIs could thus be nested inside the first NMI handler. The NMI interrupt handler should take this possibility into consideration.

Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI interrupts from inside of SMM. This behavior is implementation specific for the Pentium processor and is not part of the IA-32 architecture.

## 34.9 SMM REVISION IDENTIFIER

The SMM revision identifier field is used to indicate the version of SMM and the SMM extensions that are supported by the processor (see Figure 34-2). The SMM revision identifier is written during SMM entry and can be examined in SMRAM space at offset 7EFCH. The lower word of the SMM revision identifier refers to the version of the base SMM architecture.

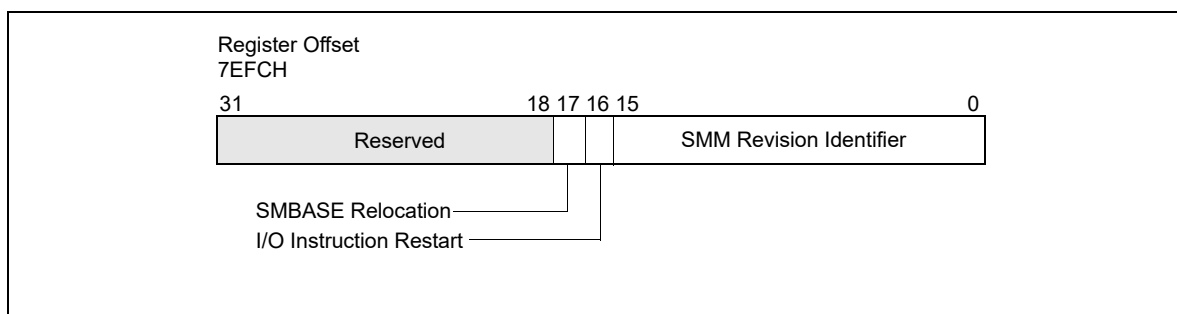


Figure 34-2. SMM Revision Identifier

The upper word of the SMM revision identifier refers to the extensions available. If the I/O instruction restart flag (bit 16) is set, the processor supports the I/O instruction restart (see Section 34.12); if the SMBASE relocation flag (bit 17) is set, SMRAM base address relocation is supported (see Section 34.11).

## 34.10 AUTO HALT RESTART

If the processor is in a HALT state (due to the prior execution of a HLT instruction) when it receives an SMI, the processor records the fact in the auto HALT restart flag in the saved processor state (see Figure 34-3). (This flag is located at offset 7F02H and bit 0 in the state save area of the SMRAM.)

If the processor sets the auto HALT restart flag upon entering SMM (indicating that the SMI occurred when the processor was in the HALT state), the SMI handler has two options:



- It can leave the auto HALT restart flag set, which instructs the RSM instruction to return program control to the HLT instruction. This option in effect causes the processor to re-enter the HALT state after handling the SMI. (This is the default operation.)
- It can clear the auto HALT restart flag, which instructs the RSM instruction to return program control to the instruction following the HLT instruction.

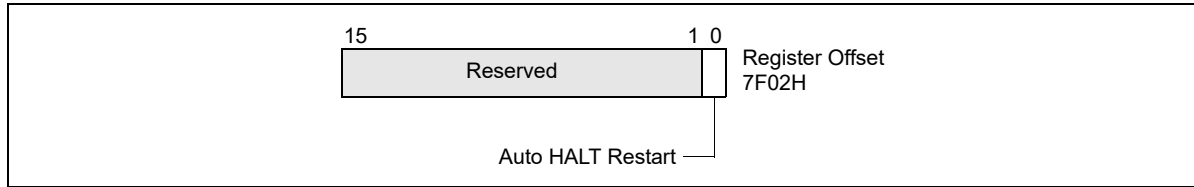


Figure 34-3. Auto HALT Restart Field

These options are summarized in Table 34-7. If the processor was not in a HALT state when the SMI was received (the auto HALT restart flag is cleared), setting the flag to 1 will cause unpredictable behavior when the RSM instruction is executed.

Table 34-7. Auto HALT Restart Flag Values

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
0	0	Returns to next instruction in interrupted program or task.
0	1	Unpredictable.
1	0	Returns to next instruction after HLT instruction.
1	1	Returns to HALT state.

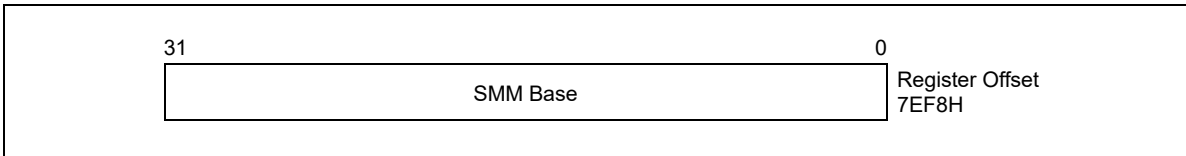
If the HLT instruction is restarted, the processor will generate a memory access to fetch the HLT instruction (if it is not in the internal cache), and execute a HLT bus transaction. This behavior results in multiple HLT bus transactions for the same HLT instruction.

### 34.10.1 Executing the HLT Instruction in SMM

The HLT instruction should not be executed during SMM, unless interrupts have been enabled by setting the IF flag in the EFLAGS register. If the processor is halted in SMM, the only event that can remove the processor from this state is a maskable hardware interrupt or a hardware reset.

## 34.11 SMBASE RELOCATION

The default base address for the SMRAM is 30000H. This value is contained in an internal processor register called the SMBASE register. The operating system or executive can relocate the SMRAM by setting the SMBASE field in the saved state map (at offset 7EF8H) to a new value (see Figure 34-4). The RSM instruction reloads the internal SMBASE register with the value in the SMBASE field each time it exits SMM. All subsequent SMI requests will use the new SMBASE value to find the starting address for the SMI handler (at SMBASE + 8000H) and the SMRAM state save area (from SMBASE + FE00H to SMBASE + FFFFH). (The processor resets the value in its internal SMBASE register to 30000H on a RESET, but does not change it on an INIT.)



**Figure 34-4. SMBASE Relocation Field**

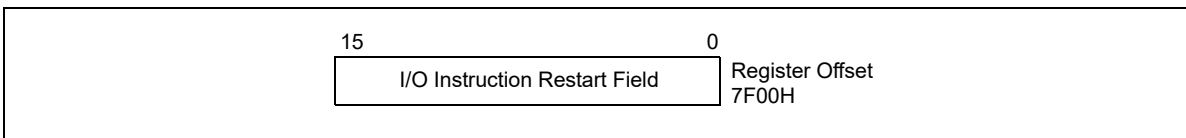
In multiple-processor systems, initialization software must adjust the SMBASE value for each processor so that the SMRAM state save areas for each processor do not overlap. (For Pentium and Intel486 processors, the SMBASE values must be aligned on a 32-KByte boundary or the processor will enter shutdown state during the execution of a RSM instruction.)

If the SMBASE relocation flag in the SMM revision identifier field is set, it indicates the ability to relocate the SMBASE (see Section 34.9).

## 34.12 I/O INSTRUCTION RESTART

If the I/O instruction restart flag in the SMM revision identifier field is set (see Section 34.9), the I/O instruction restart mechanism is present on the processor. This mechanism allows an interrupted I/O instruction to be re-executed upon returning from SMM mode. For example, if an I/O instruction is used to access a powered-down I/O device, a chip set supporting this device can intercept the access and respond by asserting SMI#. This action invokes the SMI handler to power-up the device. Upon returning from the SMI handler, the I/O instruction restart mechanism can be used to re-execute the I/O instruction that caused the SMI.

The I/O instruction restart field (at offset 7F00H in the SMM state-save area, see Figure 34-5) controls I/O instruction restart. When an RSM instruction is executed, if this field contains the value FFH, then the EIP register is modified to point to the I/O instruction that received the SMI request. The processor will then automatically re-execute the I/O instruction that the SMI trapped. (The processor saves the necessary machine state to insure that re-execution of the instruction is handled coherently.)



**Figure 34-5. I/O Instruction Restart Field**

If the I/O instruction restart field contains the value 00H when the RSM instruction is executed, then the processor begins program execution with the instruction following the I/O instruction. (When a repeat prefix is being used, the next instruction may be the next I/O instruction in the repeat loop.) Not re-executing the interrupted I/O instruction is the default behavior; the processor automatically initializes the I/O instruction restart field to 00H upon entering SMM. Table 34-8 summarizes the states of the I/O instruction restart field.

**Table 34-8. I/O Instruction Restart Field Values**

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
00H	00H	Does not re-execute trapped I/O instruction.
00H	FFH	Re-executes trapped I/O instruction.

The I/O instruction restart mechanism does not indicate the cause of the SMI. It is the responsibility of the SMI handler to examine the state of the processor to determine the cause of the SMI and to determine if an I/O instruction was interrupted and should be restarted upon exiting SMM. If an SMI interrupt is signaled on a non-I/O instruction boundary, setting the I/O instruction restart field to FFH prior to executing the RSM instruction will likely result in a program error.

### 34.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used

If an SMI interrupt is signaled while the processor is servicing an SMI interrupt that occurred on an I/O instruction boundary, the processor will service the new SMI request before restarting the originally interrupted I/O instruction. If the I/O instruction restart field is set to FFH prior to returning from the second SMI handler, the EIP will point to an address different from the originally interrupted I/O instruction, which will likely lead to a program error. To avoid this situation, the SMI handler must be able to recognize the occurrence of back-to-back SMI interrupts when I/O instruction restart is being used and insure that the handler sets the I/O instruction restart field to 00H prior to returning from the second invocation of the SMI handler.

## 34.13 SMM MULTIPLE-PROCESSOR CONSIDERATIONS

The following should be noted when designing multiple-processor systems:

- Any processor in a multiprocessor system can respond to an SMM.
- Each processor needs its own SMRAM space. This space can be in system memory or in a separate RAM.
- The SMRAMs for different processors can be overlapped in the same memory space. The only stipulation is that each processor needs its own state save area and its own dynamic data storage area. (Also, for the Pentium and Intel486 processors, the SMBASE address must be located on a 32-KByte boundary.) Code and static data can be shared among processors. Overlapping SMRAM spaces can be done more efficiently with the P6 family processors because they do not require that the SMBASE address be on a 32-KByte boundary.
- The SMI handler will need to initialize the SMBASE for each processor.
- Processors can respond to local SMIs through their SMI# pins or to SMIs received through the APIC interface. The APIC interface can distribute SMIs to different processors.
- Two or more processors can be executing in SMM at the same time.
- When operating Pentium processors in dual processing (DP) mode, the SMIACT# pin is driven only by the MRM processor and should be sampled with ADS#. For additional details, see Chapter 14 of the *Pentium Processor Family User's Manual, Volume 1*.

SMM is not re-entrant, because the SMRAM State Save Map is fixed relative to the SMBASE. If there is a need to support two or more processors in SMM mode at the same time then each processor should have dedicated SMRAM spaces. This can be done by using the SMBASE Relocation feature (see Section 34.11).

## 34.14 DEFAULT TREATMENT OF SMIS AND SMM WITH VMX OPERATION AND SMX OPERATION

Under the default treatment, the interactions of SMIs and SMM with VMX operation are few. This section details those interactions. It also explains how this treatment affects SMX operation.

### 34.14.1 Default Treatment of SMI Delivery

Ordinary SMI delivery saves processor state into SMRAM and then loads state based on architectural definitions. Under the default treatment, processors that support VMX operation perform SMI delivery as follows:

```

enter SMM;
save the following internal to the processor:
    CR4.VMXE
    an indication of whether the logical processor was in VMX operation (root or non-root)
IF the logical processor is in VMX operation
    THEN
        save current VMCS pointer internal to the processor;
        leave VMX operation;
        save VMX-critical state defined below;

```

```

FI;
IF the logical processor supports SMX operation
  THEN
    save internal to the logical processor an indication of whether the Intel® TXT private space is locked;
    IF the TXT private space is unlocked
      THEN lock the TXT private space;
    FI;
  FI;
FI;
CR4.VMXE ← 0;
perform ordinary SMI delivery:
  save processor state in SMRAM;
  set processor state to standard SMM values;1
  invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H
  are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3);

```

The pseudocode above makes reference to the saving of **VMX-critical state**. This state consists of the following: (1) SS.DPL (the current privilege level); (2) RFLAGS.VM<sup>2</sup>; (3) the state of blocking by STI and by MOV SS (see Table 24-3 in Section 24.4.2); (4) the state of virtual-NMI blocking (only if the processor is in VMX non-root operation and the “virtual NMIs” VM-execution control is 1); and (5) an indication of whether an MTF VM exit is pending (see Section 25.5.2). These data may be saved internal to the processor or in the VMCS region of the current VMCS. Processors that do not support SMI recognition while there is blocking by STI or by MOV SS need not save the state of such blocking.

If the logical processor supports the 1-setting of the “enable EPT” VM-execution control and the logical processor was in VMX non-root operation at the time of an SMI, it saves the value of that control into bit 0 of the 32-bit field at offset SMBASE + 8000H + 7EE0H (SMBASE + FEE0H; see Table 34-3).<sup>3</sup> If the logical processor was not in VMX non-root operation at the time of the SMI, it saves 0 into that bit. If the logical processor saves 1 into that bit (it was in VMX non-root operation and the “enable EPT” VM-execution control was 1), it saves the value of the EPT pointer (EPTP) into the 64-bit field at offset SMBASE + 8000H + 7ED8H (SMBASE + FED8H).

Because SMI delivery causes a logical processor to leave VMX operation, all the controls associated with VMX non-root operation are disabled in SMM and thus cannot cause VM exits while the logical processor in SMM.

### 34.14.2 Default Treatment of RSM

Ordinary execution of RSM restores processor state from SMRAM. Under the default treatment, processors that support VMX operation perform RSM as follows:

```

IF VMXE = 1 in CR4 image in SMRAM
  THEN fail and enter shutdown state;
  ELSE
    restore state normally from SMRAM;
    invalidate linear mappings and combined mappings associated with all VPIDs and all PCIDs; combined mappings are invalidated
    for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3);
    IF the logical processor supports SMX operation and the Intel® TXT private space was unlocked at the time of the last SMI (as
    saved)
      THEN unlock the TXT private space;
    FI;
  CR4.VMXE ← value stored internally;

```

1. This causes the logical processor to block INIT signals, NMIs, and SMIs.
2. Section 34.14 and Section 34.15 use 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 these registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to the lower 32 bits of the register.
3. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, SMI functions as the “enable EPT” VM-execution control were 0. See Section 24.6.2.

IF internal storage indicates that the logical processor  
had been in VMX operation (root or non-root)

THEN

enter VMX operation (root or non-root);

restore VMX-critical state as defined in Section 34.14.1;

set to their fixed values any bits in CR0 and CR4 whose values must be fixed in VMX operation (see Section 23.8);<sup>1</sup>

IF RFLAGS.VM = 0 AND (in VMX root operation OR the “unrestricted guest” VM-execution control is 0)<sup>2</sup>

THEN

CS.RPL ← SS.DPL;

SS.RPL ← SS.DPL;

FI;

restore current VMCS pointer;

FI;

leave SMM;

IF logical processor will be in VMX operation or in SMX operation after RSM

THEN block A20M and leave A20M mode;

FI;

FI;

RSM unblocks SMIs. It restores the state of blocking by NMI (see Table 24-3 in Section 24.4.2) as follows:

- If the RSM is not to VMX non-root operation or if the “virtual NMIs” VM-execution control will be 0, the state of NMI blocking is restored normally.
- If the RSM is to VMX non-root operation and the “virtual NMIs” VM-execution control will be 1, NMIs are not blocked after RSM. The state of virtual-NMI blocking is restored as part of VMX-critical state.

INIT signals are blocked after RSM if and only if the logical processor will be in VMX root operation.

If RSM returns a logical processor to VMX non-root operation, it re-establishes the controls associated with the current VMCS. If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs immediately after RSM if the enabling conditions apply. The same is true for the “NMI-window exiting” VM-execution control. Such VM exits occur with their normal priority. See Section 25.2.

If an MTF VM exit was pending at the time of the previous SMI, an MTF VM exit is pending on the instruction boundary following execution of RSM. The following items detail the treatment of MTF VM exits that may be pending following RSM:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these MTF VM exits. These MTF VM exits take priority over debug-trap exceptions and lower priority events.
- These MTF VM exits wake the logical processor if RSM caused the logical processor to enter the HLT state (see Section 34.10). They do not occur if the logical processor just entered the shutdown state.

### 34.14.3 Protection of CR4.VMXE in SMM

Under the default treatment, CR4.VMXE is treated as a reserved bit while a logical processor is in SMM. Any attempt by software running in SMM to set this bit causes a general-protection exception. In addition, software cannot use VMX instructions or enter VMX operation while in SMM.

### 34.14.4 VMXOFF and SMI Unblocking

The VMXOFF instruction can be executed only with the default treatment (see Section 34.15.1) and only outside SMM. If SMIs are blocked when VMXOFF is executed, VMXOFF unblocks them unless

1. If the RSM is to VMX non-root operation and both the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls will be 1, CR0.PE and CR0.PG retain the values that were loaded from SMRAM regardless of what is reported in the capability MSR IA32\_VMX\_CRO\_FIXED0.
2. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

IA32\_SMM\_MONITOR\_CTL[bit 2] is 1 (see Section 34.15.5 for details regarding this MSR).<sup>1</sup> Section 34.15.7 identifies a case in which SMIs may be blocked when VMXOFF is executed.

Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine whether this is allowed.

## 34.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM

Dual-monitor treatment is activated through the cooperation of the **executive monitor** (the VMM that operates outside of SMM to provide basic virtualization) and the **SMM-transfer monitor (STM)** (the VMM that operates inside SMM—while in VMX operation—to support system-management functions). Control is transferred to the STM through VM exits; VM entries are used to return from SMM.

The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

### 34.15.1 Dual-Monitor Treatment Overview

The dual-monitor treatment uses an executive monitor and an SMM-transfer monitor (STM). Transitions from the executive monitor or its guests to the STM are called **SMM VM exits** and are discussed in Section 34.15.2. SMM VM exits are caused by SMIs as well as executions of VMCALL in VMX root operation. The latter allow the executive monitor to call the STM for service.

The STM runs in VMX root operation and uses VMX instructions to establish a VMCS and perform VM entries to its own guests. This is done all inside SMM (see Section 34.15.3). The STM returns from SMM, not by using the RSM instruction, but by using a VM entry that returns from SMM. Such VM entries are described in Section 34.15.4.

Initially, there is no STM and the default treatment (Section 34.14) is used. The dual-monitor treatment is not used until it is enabled and activated. The steps to do this are described in Section 34.15.5 and Section 34.15.6.

It is not possible to leave VMX operation under the dual-monitor treatment; VMXOFF will fail if executed. The dual-monitor treatment must be deactivated first. The STM deactivates dual-monitor treatment using a VM entry that returns from SMM with the “deactivate dual-monitor treatment” VM-entry control set to 1 (see Section 34.15.7).

The executive monitor configures any VMCS that it uses for VM exits to the executive monitor. SMM VM exits, which transfer control to the STM, use a different VMCS. Under the dual-monitor treatment, each logical processor uses a separate VMCS called the **SMM-transfer VMCS**. When the dual-monitor treatment is active, the logical processor maintains another VMCS pointer called the **SMM-transfer VMCS pointer**. The SMM-transfer VMCS pointer is established when the dual-monitor treatment is activated.

### 34.15.2 SMM VM Exits

An SMM VM exit is a VM exit that begins outside SMM and that ends in SMM.

Unlike other VM exits, SMM VM exits can begin in VMX root operation. SMM VM exits result from the arrival of an SMI outside SMM or from execution of VMCALL in VMX root operation outside SMM. Execution of VMCALL in VMX root operation causes an SMM VM exit only if the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR (see Section 34.15.5).

Execution of VMCALL in VMX root operation causes an SMM VM exit even under the default treatment. This SMM VM exit activates the dual-monitor treatment (see Section 34.15.6).

Differences between SMM VM exits and other VM exits are detailed in Sections 34.15.2.1 through 34.15.2.5. Differences between SMM VM exits that activate the dual-monitor treatment and other SMM VM exits are described in Section 34.15.6.

---

1. Setting IA32\_SMM\_MONITOR\_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s valid bit (bit 0).

### 34.15.2.1 Architectural State Before a VM Exit

System-management interrupts (SMIs) that cause SMM VM exits always do so directly. They do not save state to SMRAM as they do under the default treatment.

### 34.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers

SMM VM exits begin by performing the following steps:

1. The executive-VMCS pointer field in the SMM-transfer VMCS is loaded as follows:
  - If the SMM VM exit commenced in VMX non-root operation, it receives the current-VMCS pointer.
  - If the SMM VM exit commenced in VMX root operation, it receives the VMXON pointer.
2. The current-VMCS pointer is loaded with the value of the SMM-transfer VMCS pointer.

The last step ensures that the current VMCS is the SMM-transfer VMCS. VM-exit information is recorded in that VMCS, and VM-entry control fields in that VMCS are updated. State is saved into the guest-state area of that VMCS. The VM-exit controls and host-state area of that VMCS determine how the VM exit operates.

### 34.15.2.3 Recording VM-Exit Information

SMM VM exits differ from other VM exit with regard to the way they record VM-exit information. The differences follow.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. The field is loaded with the reason for the SMM VM exit: I/O SMI (an SMI arrived immediately after retirement of an I/O instruction), other SMI, or VMCALL. See Appendix C, “VMX Basic Exit Reasons”.
  - SMM VM exits are the only VM exits that may occur in VMX root operation. Because the SMM-transfer monitor may need to know whether it was invoked from VMX root or VMX non-root operation, this information is stored in bit 29 of the exit-reason field (see Table 24-14 in Section 24.9.1). The bit is set by SMM VM exits from VMX root operation.
  - If the SMM VM exit occurred in VMX non-root operation and an MTF VM exit was pending, bit 28 of the exit-reason field is set; otherwise, it is cleared.
  - Bits 27:16 and bits 31:30 are cleared.
- **Exit qualification.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, the exit qualification contains information about the I/O instruction that retired immediately before the SMI. It has the format given in Table 34-9.

**Table 34-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction**

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte  Other values not used.
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)

**Table 34-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction (Contd.)**

Bit Position(s)	Contents
15:7	Reserved (cleared to 0)
31:16	Port number (as specified in the I/O instruction)
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

- **Guest linear address.** This field is used for VM exits due to SMIs that arrive immediately after the retirement of an INS or OUTS instruction for which the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) is usable. The field receives the value of the linear address generated by ES:(E)DI (for INS) or segment:(E)SI (for OUTS; the default segment is DS but can be overridden by a segment override prefix) at the time the instruction started. If the relevant segment is not usable, the value is undefined. On processors that support Intel 64 architecture, bits 63:32 are clear if the logical processor was not in 64-bit mode before the VM exit.
- **I/O RCX, I/O RSI, I/O RDI, and I/O RIP.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, these fields receive the values that were in RCX, RSI, RDI, and RIP, respectively, before the I/O instruction executed. Thus, the value saved for I/O RIP addresses the I/O instruction.

#### 34.15.2.4 Saving Guest State

SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area.

The value of the VMX-preemption timer is saved into the corresponding field in the guest-state area if the “save VMX-preemption timer value” VM-exit control is 1. That field becomes undefined if, in addition, either the SMM VM exit is from VMX root operation or the SMM VM exit is from VMX non-root operation and the “activate VMX-preemption timer” VM-execution control is 0.

#### 34.15.2.5 Updating Non-Register State

SMM VM exits affect the non-register state of a logical processor as follows:

- SMM VM exits cause non-maskable interrupts (NMIs) to be blocked; they may be unblocked through execution of IRET or through a VM entry (depending on the value loaded for the interruptibility state and the setting of the “virtual NMIs” VM-execution control).
- SMM VM exits cause SMIs to be blocked; they may be unblocked by a VM entry that returns from SMM (see Section 34.15.4).

SMM VM exits invalidate linear mappings and combined mappings associated with VPID 0000H for all PCIDs. Combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3). (Ordinary VM exits are not required to perform such invalidation if the “enable VPID” VM-execution control is 1; see Section 27.5.5.)

### 34.15.3 Operation of the SMM-Transfer Monitor

Once invoked, the SMM-transfer monitor (STM) is in VMX root operation and can use VMX instructions to configure VMCSs and to cause VM entries to virtual machines supported by those structures. As noted in Section 34.15.1, the VMXOFF instruction cannot be used under the dual-monitor treatment and thus cannot be used by the STM.

The RSM instruction also cannot be used under the dual-monitor treatment. As noted in Section 25.1.3, it causes a VM exit if executed in SMM in VMX non-root operation. If executed in VMX root operation, it causes an invalid-opcode exception. The STM uses VM entries to return from SMM (see Section 34.15.4).



### 34.15.4 VM Entries that Return from SMM

The SMM-transfer monitor (STM) returns from SMM using a VM entry with the “entry to SMM” VM-entry control clear. VM entries that return from SMM reverse the effects of an SMM VM exit (see Section 34.15.2).

VM entries that return from SMM may differ from other VM entries in that they do not necessarily enter VMX non-root operation. If the executive-VMCS pointer field in the current VMCS contains the VMXON pointer, the logical processor remains in VMX root operation after VM entry.

For differences between VM entries that return from SMM and other VM entries see Sections 34.15.4.1 through 34.15.4.10.

#### 34.15.4.1 Checks on the Executive-VMCS Pointer Field

VM entries that return from SMM perform the following checks on the executive-VMCS pointer field in the current VMCS:

- Bits 11:0 must be 0.
- The pointer must not set any bits beyond the processor’s physical-address width.<sup>1,2</sup>
- The 32 bits located in memory referenced by the physical address in the pointer must contain the processor’s VMCS revision identifier (see Section 24.2).

The checks above are performed before the checks described in Section 34.15.4.2 and before any of the following checks:

- If the “deactivate dual-monitor treatment” VM-entry control is 0 and the executive-VMCS pointer field does not contain the VMXON pointer, the launch state of the executive VMCS (the VMCS referenced by the executive-VMCS pointer field) must be launched (see Section 24.11.3).
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the executive-VMCS pointer field must contain the VMXON pointer (see Section 34.15.7).<sup>3</sup>

#### 34.15.4.2 Checks on VM-Execution Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-execution control fields specified in Section 26.2.1.1. They do not apply the checks to the current VMCS. Instead, VM-entry behavior depends on whether the executive-VMCS pointer field contains the VMXON pointer:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are not performed at all.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the checks are performed on the VM-execution control fields in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS). These checks are performed after checking the executive-VMCS pointer field itself (for proper alignment).

Other VM entries ensure that, if “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-exit control is also 0. This check is not performed by VM entries that return from SMM.

#### 34.15.4.3 Checks on VM-Entry Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-entry control fields specified in Section 26.2.1.3.

- 
1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
  2. If IA32\_VMX\_BASIC[48] is read as 1, this pointer must not set any bits in the range 63:32; see Appendix A.1.
  3. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

Specifically, if the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the VM-entry interruption-information field must not indicate injection of a pending MTF VM exit (see Section 26.5.2). Specifically, the following cannot all be true for that field:

- the valid bit (bit 31) is 1
- the interruption type (bits 10:8) is 7 (other event); and
- the vector (bits 7:0) is 0 (pending MTF VM exit).

#### 34.15.4.4 Checks on the Guest State Area

Section 26.3.1 specifies checks performed on fields in the guest-state area of the VMCS. Some of these checks are conditioned on the settings of certain VM-execution controls (e.g., “virtual NMIs” or “unrestricted guest”). VM entries that return from SMM modify these checks based on whether the executive-VMCS pointer field contains the VMXON pointer:<sup>1</sup>

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are performed as all relevant VM-execution controls were 0. (As a result, some checks may not be performed at all.)
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), this check is performed based on the settings of the VM-execution controls in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS).

For VM entries that return from SMM, the activity-state field must not indicate the wait-for-SIPI state if the executive-VMCS pointer field contains the VMXON pointer (the VM entry is to VMX root operation).

#### 34.15.4.5 Loading Guest State

VM entries that return from SMM load the SMBASE register from the SMBASE field.

VM entries that return from SMM invalidate linear mappings and combined mappings associated with all VPIDs. Combined mappings are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3). (Ordinary VM entries are required to perform such invalidation only for VPID 0000H and are not required to do even that if the “enable VPID” VM-execution control is 1; see Section 26.3.2.5.)

#### 34.15.4.6 VMX-Preemption Timer

A VM entry that returns from SMM activates the VMX-preemption timer only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation) and the “activate VMX-preemption timer” VM-execution control is 1 in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field). In this case, VM entry starts the VMX-preemption timer with the value in the VMX-preemption timer-value field in the current VMCS.

#### 34.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers

Successful VM entries (returning from SMM) load the SMM-transfer VMCS pointer with the current-VMCS pointer. Following this, they load the current-VMCS pointer from a field in the current VMCS:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the current-VMCS pointer is loaded from the VMCS-link pointer field.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the current-VMCS pointer is loaded with the value of the executive-VMCS pointer field.

If the VM entry successfully enters VMX non-root operation, the VM-execution controls in effect after the VM entry are those from the new current VMCS. This includes any structures external to the VMCS referenced by VM-execution control fields.

---

1. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

The updating of these VMCS pointers occurs before event injection. Event injection is determined, however, by the VM-entry control fields in the VMCS that was current when the VM entry commenced.

#### 34.15.4.8 VM Exits Induced by VM Entry

Section 26.5.1.2 describes how the event-delivery process invoked by event injection may lead to a VM exit. Section 26.6.3 to Section 26.6.7 describe other situations that may cause a VM exit to occur immediately after a VM entry.

Whether these VM exits occur is determined by the VM-execution control fields in the current VMCS. For VM entries that return from SMM, they can occur only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation).

In this case, determination is based on the VM-execution control fields in the VMCS that is current after the VM entry. This is the VMCS referenced by the value of the executive-VMCS pointer field at the time of the VM entry (see Section 34.15.4.7). This VMCS also controls the delivery of such VM exits. Thus, VM exits induced by a VM entry returning from SMM are to the executive monitor and not to the STM.

#### 34.15.4.9 SMI Blocking

VM entries that return from SMM determine the blocking of system-management interrupts (SMIs) as follows:

- If the “deactivate dual-monitor treatment” VM-entry control is 0, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the blocking of SMIs depends on whether the logical processor is in SMX operation:<sup>1</sup>
  - If the logical processor is in SMX operation, SMIs are blocked after VM entry.
  - If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

VM entries that return from SMM and that do not deactivate the dual-monitor treatment may leave SMIs blocked. This feature exists to allow the STM to invoke functionality outside of SMM without unblocking SMIs.

#### 34.15.4.10 Failures of VM Entries That Return from SMM

Section 26.7 describes the treatment of VM entries that fail during or after loading guest state. Such failures record information in the VM-exit information fields and load processor state as would be done on a VM exit. The VMCS used is the one that was current before the VM entry commenced. Control is thus transferred to the STM and the logical processor remains in SMM.

### 34.15.5 Enabling the Dual-Monitor Treatment

Code and data for the SMM-transfer monitor (STM) reside in a region of SMRAM called the **monitor segment** (MSEG). Code running in SMM determines the location of MSEG and establishes its content. This code is also responsible for enabling the dual-monitor treatment.

SMM code enables the dual-monitor treatment and specifies the location of MSEG by writing to the IA32\_SMM\_MONITOR\_CTL MSR (index 9BH). The MSR has the following format:

- Bit 0 is the register’s valid bit. The STM may be invoked using VMCALL only if this bit is 1. Because VMCALL is used to activate the dual-monitor treatment (see Section 34.15.6), the dual-monitor treatment cannot be activated if the bit is 0. This bit is cleared when the logical processor is reset.
- Bit 1 is reserved.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.

- Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant). See Section 34.14.4. Certain leaf functions of the GETSEC instruction clear this bit (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*).
- Bits 11:3 are reserved.
- Bits 31:12 contain a value that, when shifted left 12 bits, is the physical address of MSEG (the MSEG base address).
- Bits 63:32 are reserved.

The following items detail use of this MSR:

- The IA32\_SMM\_MONITOR\_CTL MSR is supported only on processors that support the dual-monitor treatment.<sup>1</sup> On other processors, accesses to the MSR using RDMSR or WRMSR generate a general-protection fault (#GP(0)).
- A write to the IA32\_SMM\_MONITOR\_CTL MSR using WRMSR generates a general-protection fault (#GP(0)) if executed outside of SMM or if an attempt is made to set any reserved bit. An attempt to write to the IA32\_SMM\_MONITOR\_CTL MSR fails if made as part of a VM exit that does not end in SMM or part of a VM entry that does not begin in SMM.
- Reads from the IA32\_SMM\_MONITOR\_CTL MSR using RDMSR are allowed any time RDMSR is allowed. The MSR may be read as part of any VM exit.
- The dual-monitor treatment can be activated only if the valid bit in the MSR is set to 1.

The 32 bytes located at the MSEG base address are called the **MSEG header**. The format of the MSEG header is given in Table 34-10 (each field is 32 bits).

**Table 34-10. Format of MSEG Header**

Byte Offset	Field
0	MSEG-header revision identifier
4	SMM-transfer monitor features
8	GDTR limit
12	GDTR base offset
16	CS selector
20	EIP offset
24	ESP offset
28	CR3 offset

To ensure proper behavior in VMX operation, software should maintain the MSEG header in writeback cacheable memory. Future implementations may allow or require a different memory type.<sup>2</sup> Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1).

SMM code should enable the dual-monitor treatment (by setting the valid bit in IA32\_SMM\_MONITOR\_CTL MSR) only after establishing the content of the MSEG header as follows:

1. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.
2. Alternatively, software may map the MSEG header with the UC memory type; this may be necessary, depending on how memory is organized. 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.

- Bytes 3:0 contain the **MSEG revision identifier**. Different processors may use different MSEG revision identifiers. These identifiers enable software to avoid using an MSEG header formatted for one processor on a processor that uses a different format. Software can discover the MSEG revision identifier that a processor uses by reading the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6).
- Bytes 7:4 contain the **SMM-transfer monitor features** field. Bits 31:1 of this field are reserved and must be zero. Bit 0 of the field is the **IA-32e mode SMM feature bit**. It indicates whether the logical processor will be in IA-32e mode after the STM is activated (see Section 34.15.6).
- Bytes 31:8 contain fields that determine how processor state is loaded when the STM is activated (see Section 34.15.6.5). SMM code should establish these fields so that activating of the STM invokes the STM's initialization code.

### 34.15.6 Activating the Dual-Monitor Treatment

The dual-monitor treatment may be enabled by SMM code as described in Section 34.15.5. The dual-monitor treatment is activated only if it is enabled and only by the executive monitor. The executive monitor activates the dual-monitor treatment by executing VMCALL in VMX root operation.

When VMCALL activates the dual-monitor treatment, it causes an SMM VM exit. Differences between this SMM VM exit and other SMM VM exits are discussed in Sections 34.15.6.1 through 34.15.6.6. See also “VMCALL—Call to VM Monitor” in Chapter 30.

#### 34.15.6.1 Initial Checks

An execution of VMCALL attempts to activate the dual-monitor treatment if (1) the processor supports the dual-monitor treatment;<sup>1</sup> (2) the logical processor is in VMX root operation; (3) the logical processor is outside SMM and the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR; (4) the logical processor is not in virtual-8086 mode and not in compatibility mode; (5) CPL = 0; and (6) the dual-monitor treatment is not active.

Such an execution of VMCALL begins with some initial checks. These checks are performed before updating the current-VMCS pointer and the executive-VMCS pointer field (see Section 34.15.2.2).

The VMCS that manages SMM VM exit caused by this VMCALL is the current VMCS established by the executive monitor. The VMCALL performs the following checks on the current VMCS in the order indicated:

1. There must be a current VMCS pointer.
2. The launch state of the current VMCS must be clear.
3. Reserved bits in the VM-exit controls in the current VMCS must be set properly. Software may consult the VMX capability MSR IA32\_VMX\_EXIT\_CTLS to determine the proper settings (see Appendix A.4).

If any of these checks fail, subsequent checks are skipped and VMCALL fails. If all these checks succeed, the logical processor uses the IA32\_SMM\_MONITOR\_CTL MSR to determine the base address of MSEG. The following checks are performed in the order indicated:

1. The logical processor reads the 32 bits at the base of MSEG and compares them to the processor's MSEG revision identifier.
2. The logical processor reads the SMM-transfer monitor features field:
  - Bit 0 of the field is the IA-32e mode SMM feature bit, and it indicates whether the logical processor will be in IA-32e mode after the SMM-transfer monitor (STM) is activated.
    - If the VMCALL is executed on a processor that does not support Intel 64 architecture, the IA-32e mode SMM feature bit must be 0.
    - If the VMCALL is executed in 64-bit mode, the IA-32e mode SMM feature bit must be 1.
  - Bits 31:1 of this field are currently reserved and must be zero.

If any of these checks fail, subsequent checks are skipped and the VMCALL fails.

- 
1. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.

### 34.15.6.2 Updating the Current-VMCS and Executive-VMCS Pointers

Before performing the steps in Section 34.15.2.2, SMM VM exits that activate the dual-monitor treatment begin by loading the SMM-transfer VMCS pointer with the value of the current-VMCS pointer.

### 34.15.6.3 Saving Guest State

As noted in Section 34.15.2.4, SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area. While this is true also for SMM VM exits that activate the dual-monitor treatment, the VMCS used for those VM exits exists outside SMRAM.

The SMM-transfer monitor (STM) can also discover the current value of the SMBASE register by using the RDMSR instruction to read the IA32\_SMBASE MSR (MSR address 9EH). The following items detail use of this MSR:

- The MSR is supported only if IA32\_VMX\_MISC[15] = 1 (see Appendix A.6).
- A write to the IA32\_SMBASE MSR using WRMSR generates a general-protection fault (#GP(0)). An attempt to write to the IA32\_SMBASE MSR fails if made as part of a VM exit or part of a VM entry.
- A read from the IA32\_SMBASE MSR using RDMSR generates a general-protection fault (#GP(0)) if executed outside of SMM. An attempt to read from the IA32\_SMBASE MSR fails if made as part of a VM exit that does not end in SMM.

### 34.15.6.4 Saving MSRs

The VM-exit MSR-store area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are saved into that area.

### 34.15.6.5 Loading Host State

The VMCS that is current during an SMM VM exit that activates the dual-monitor treatment was established by the executive monitor. It does not contain the VM-exit controls and host state required to initialize the STM. For this reason, such SMM VM exits do not load processor state as described in Section 27.5. Instead, state is set to fixed values or loaded based on the content of the MSEG header (see Table 34-10):

- CR0 is set to as follows:
  - PG, NE, ET, MP, and PE are all set to 1.
  - CD and NW are left unchanged.
  - All other bits are cleared to 0.
- CR3 is set as follows:
  - Bits 63:32 are cleared on processors that support IA-32e mode.
  - Bits 31:12 are set to bits 31:12 of the sum of the MSEG base address and the CR3-offset field in the MSEG header.
  - Bits 11:5 and bits 2:0 are cleared (the corresponding bits in the CR3-offset field in the MSEG header are ignored).
  - Bits 4:3 are set to bits 4:3 of the CR3-offset field in the MSEG header.
- CR4 is set as follows:
  - MCE, PGE, and PCIDE are cleared.
  - PAE is set to the value of the IA-32e mode SMM feature bit.
  - If the IA-32e mode SMM feature bit is clear, PSE is set to 1 if supported by the processor; if the bit is set, PSE is cleared.
  - All other bits are unchanged.
- DR7 is set to 400H.
- The IA32\_DEBUGCTL MSR is cleared to 00000000\_00000000H.

- The registers CS, SS, DS, ES, FS, and GS are loaded as follows:
  - All registers are usable.
  - CS.selector is loaded from the corresponding field in the MSEG header (the high 16 bits are ignored), with bits 2:0 cleared to 0. If the result is 0000H, CS.selector is set to 0008H.
  - The selectors for SS, DS, ES, FS, and GS are set to CS.selector+0008H. If the result is 0000H (if the CS selector was FFF8H), these selectors are instead set to 0008H.
  - The base addresses of all registers are cleared to zero.
  - The segment limits for all registers are set to FFFFFFFFH.
  - The AR bytes for the registers are set as follows:
    - CS.Type is set to 11 (execute/read, accessed, non-conforming code segment).
    - For SS, DS, ES, FS, and GS, the Type is set to 3 (read/write, accessed, expand-up data segment).
    - The S bits for all registers are set to 1.
    - The DPL for each register is set to 0.
    - The P bits for all registers are set to 1.
    - On processors that support Intel 64 architecture, CS.L is loaded with the value of the IA-32e mode SMM feature bit.
    - CS.D is loaded with the inverse of the value of the IA-32e mode SMM feature bit.
    - For each of SS, DS, ES, FS, and GS, the D/B bit is set to 1.
    - The G bits for all registers are set to 1.
- LDTR is unusable. The LDTR selector is cleared to 0000H, and the register is otherwise undefined (although the base address is always canonical)
- GDTR.base is set to the sum of the MSEG base address and the GDTR base-offset field in the MSEG header (bits 63:32 are always cleared on processors that support IA-32e mode). GDTR.limit is set to the corresponding field in the MSEG header (the high 16 bits are ignored).
- IDTR.base is unchanged. IDTR.limit is cleared to 0000H.
- RIP is set to the sum of the MSEG base address and the value of the RIP-offset field in the MSEG header (bits 63:32 are always cleared on logical processors that support IA-32e mode).
- RSP is set to the sum of the MSEG base address and the value of the RSP-offset field in the MSEG header (bits 63:32 are always cleared on logical processor that supports IA-32e mode).
- RFLAGS is cleared, except bit 1, which is always set.
- The logical processor is left in the active state.
- Event blocking after the SMM VM exit is as follows:
  - There is no blocking by STI or by MOV SS.
  - There is blocking by non-maskable interrupts (NMIs) and by SMIs.
- There are no pending debug exceptions after the SMM VM exit.
- For processors that support IA-32e mode, the IA32\_EFER MSR is modified so that LME and LMA both contain the value of the IA-32e mode SMM feature bit.

If any of CR3[63:5], CR4.PAE, CR4.PSE, or IA32\_EFER.LMA is changing, the TLBs are updated so that, after VM exit, the logical processor does not use translations that were cached before the transition. This is not necessary for changes that would not affect paging due to the settings of other bits (for example, changes to CR4.PSE if IA32\_EFER.LMA was 1 before and after the transition).

### 34.15.6.6 Loading MSRs

The VM-exit MSR-load area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are loaded from that area.

### 34.15.7 Deactivating the Dual-Monitor Treatment

The SMM-transfer monitor may deactivate the dual-monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 34.14). It does this by executing a VM entry with the “deactivate dual-monitor treatment” VM-entry control set to 1.

As noted in Section 26.2.1.3 and Section 34.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the “entry to SMM” VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 34.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:<sup>1</sup>

- If the logical processor is in SMX operation, SMIs are blocked after VM entry. SMIs may later be unblocked by the VMXOFF instruction (see Section 34.14.4) or by certain leaf functions of the GETSEC instruction (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*).
- If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

## 34.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT

On processors that support processor extended states using XSAVE/XRSTOR (see Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*), the processor does not save any XSAVE/XRSTOR related state on an SMI. It is the responsibility of the SMI handler code to properly preserve the state information (including CR4.OSXSAVE, XCR0, and possibly processor extended states using XSAVE/XRSTOR). Therefore, the SMI handler must follow the rules described in Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## 34.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT

This section describes enhancement of system management features that apply only to the 4th generation Intel Core processors. These features are model-specific. BIOS and SMM handler must use CPUID to enumerate DisplayFamily\_DisplayModel signature when programming with these interfaces.

### 34.17.1 SMM Handler Code Access Control

The BIOS may choose to restrict the address ranges of code that SMM handler executes. When SMM handler code execution check is enabled, an attempt by the SMM handler to execute outside the ranges specified by SMRR (see Section 34.4.2.1) will cause the assertion of an unrecoverable machine check exception (MCE).

The interface to enable SMM handler code access check resides in a per-package scope model-specific register MSR\_SMM\_FEATURE\_CONTROL at address 4E0H. An attempt to access MSR\_SMM\_FEATURE\_CONTROL outside of SMM will cause a #GP. Writes to MSR\_SMM\_FEATURE\_CONTROL is further protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_FEATURE\_CONTROL and MSR\_SMM\_MCA\_CAP are described in Table 2-28 in Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.



### 34.17.2 SMI Delivery Delay Reporting

Entry into the system management mode occurs at instruction boundary. In situations where a logical processor is executing an instruction involving a long flow of internal operations, servicing an SMI by that logical processor will be delayed. Delayed servicing of SMI of each logical processor due to executing long flows of internal operation in a physical processor can be queried via a package-scope register MSR\_SMM\_DELAYED at address 4E2H.

The interface to enable reporting of SMI delivery delay due to long internal flows resides in a per-package scope model-specific register MSR\_SMM\_DELAYED. An attempt to access MSR\_SMM\_DELAYED outside of SMM will cause a #GP. Availability to MSR\_SMM\_DELAYED is protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_DELAYED and MSR\_SMM\_MCA\_CAP are described in Table 2-28 in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

### 34.17.3 Blocked SMI Reporting

A logical processor may have entered into a state and blocked from servicing other interrupts (including SMI). Logical processors in a physical processor that are blocked in serving SMI can be queried in a package-scope register MSR\_SMM\_BLOCKED at address 4E3H. An attempt to access MSR\_SMM\_BLOCKED outside of SMM will cause a #GP.

Details of the interface of MSR\_SMM\_BLOCKED is described in Table 2-28 in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

## 18. Updates to Chapter 35, Volume 3C

Change bars show changes to Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Change to chapter: Update to Table 35-1 "COFI Type for Branch Instructions". Update to Table 35-22 "FUP Packet Definition".

# 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	INT1, INT3, INT <i>n</i> , INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

### 35.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two sub-categories:

- **Conditional Branch (Jcc, J\*CXZ) and LOOP**

To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction.

Jcc, J\*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**

There is no trace output required for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet, though TIP.PGD and TIP.PGE packets can be generated by unconditional direct jumps that toggle Intel PT enables (see Section 35.2.5).

### 35.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 35.3.1.1).

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**

As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**

When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

- **RET Compression**

A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with “corresponding” defined

as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be “compressed”. In this case, only a single TNT bit of “taken” is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed in a given logical processor. For details, see “Indirect Transfer Compression for Returns (RET)” in Section 35.4.2.2.

### 35.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are “far transfers”. This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 35-23 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. Support for PTWRITE is introduced in Intel® Atom™ processors based on the Goldmont Plus microarchitecture.

## 35.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

- Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.
- The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.
- The last and deepest core C-state achieved during a sleep session.
- The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32\_RTIT\_CTL.PwrEvtEn[4]. Support for Power Event Tracing is introduced in Intel® Atom™ processors based on the Goldmont Plus microarchitecture.

## 35.2.4 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

### 35.2.4.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 35.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 35.2.5.1 for details.

### 35.2.4.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 35.3.5, "Context Switch Consideration").

To trace for only a single CR3 value, software can write that value to the IA32\_RTIT\_CR3\_MATCH MSR, and set IA32\_RTIT\_CTL.CR3Filter. When CR3 value does not match IA32\_RTIT\_CR3\_MATCH and IA32\_RTIT\_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 35.2.5.3 for details. When CR3 does match IA32\_RTIT\_CR3\_MATCH (or when IA32\_RTIT\_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32\_RTIT\_CR3\_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32\_RTIT\_CR3\_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32\_RTIT\_CTL.OS = 1). If not, no PIP packets will be seen.

### 35.2.4.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1. Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn\_CFG fields in the IA32\_RTIT\_CTL MSR (Section 35.2.7.2), where the digit 'n' is a zero-based number that selects which address range is being configured. Each ADDRn\_CFG field configures the use of the register pair IA32\_RTIT\_ADDRn\_A and IA32\_RTIT\_ADDRn\_B (Section 35.2.7.5). IA32\_RTIT\_ADDRn\_A defines the base and IA32\_RTIT\_ADDRn\_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32\_RTIT\_ADDRn\_A, IA32\_RTIT\_ADDRn\_B]. There can be multiple such ranges, software can query CPUID (Section 35.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn\_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn\_CFG is set to enable IP filtering (see Section 35.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 35.2.5.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 35.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 35.3.1).

Note that some packets, such as MTC (Section 35.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 35.4.1.

### TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32\_RTIT\_CTL.ADDRn\_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32\_RTIT\_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 35.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.

To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32\_RTIT\_CTL.TraceEn before the IA32\_RTIT\_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32\_RTIT\_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 35.2.6.2.

### IP Filtering Example

The following table gives an example of IP filtering behavior. Assume that IA32\_RTIT\_ADDRn\_A = the IP of RangeBase, and that IA32\_RTIT\_ADDRn\_B = the IP of RangeLimit, while IA32\_RTIT\_CTL.ADDRn\_CFG = 0x1 (enable ADDRn range as a FilterEn range).

**Table 35-2. IP Filtering Packet Example**

Code Flow	Packets
<pre> Bar:     jmp RangeBase // jump into filter range RangeBase:     jcc Foo // not taken     add eax, 1 Foo:     jmp RangeLimit+1 // jump out of filter range RangeLimit:     nop     jcc Bar                     </pre>	<pre> TIP.PGE(RangeBase) TNT(0) TIP.PGD(RangeLimit+1)                     </pre>

### IP Filtering and TraceStop

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.



## 35.2.5 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (**PacketEn**) is set. **PacketEn** is an internal state maintained in hardware in response to software configurable enable controls, **PacketEn** is not visible to software directly. The relationship of **PacketEn** to the software-visible controls in the configuration MSRs is described in this section.

### 35.2.5.1 Packet Enable (**PacketEn**)

When **PacketEn** is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. **PacketEn** is composed of other states according to this relationship:

$$\text{PacketEn} \leftarrow \text{TriggerEn} \text{ AND } \text{ContextEn} \text{ AND } \text{FilterEn} \text{ AND } \text{BranchEn}$$

These constituent controls are detailed in the following subsections.

**PacketEn** ultimately determines when the processor is tracing. When **PacketEn** is set, all control flow packets are enabled. When **PacketEn** is clear, no control flow packets are generated, though other packets (timing and book-keeping packets) may still be sent. See Section 35.2.6 for details of **PacketEn** and packet generation.

Note that, on processors that do not support IP filtering (i.e., `CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0`), **FilterEn** is treated as always set.

### 35.2.5.2 Trigger Enable (**TriggerEn**)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. **TriggerEn** is set when `IA32_RTIT_CTL.TraceEn` is set, and cleared by any of the following conditions:

- `TraceEn` is cleared by software.
- A `TraceStop` condition is encountered and `IA32_RTIT_STATUS.Stopped` is set.
- `IA32_RTIT_STATUS.Error` is set due to an operational error (see Section 35.3.9).

Software can discover the current **TriggerEn** value by reading the `IA32_RTIT_STATUS.TriggerEn` bit. When **TriggerEn** is clear, tracing is inactive and no packets are generated.

### 35.2.5.3 Context Enable (**ContextEn**)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with `CPL = 0` code is not being traced (`IA32_RTIT_CTL.OS = 0`), then **ContextEn** will be 0 when the processor is in `CPL0`.

Software can discover the current **ContextEn** value by reading the `IA32_RTIT_STATUS.ContextEn` bit. **ContextEn** is defined as follows:

$$\begin{aligned} \text{ContextEn} = & !((\text{IA32\_RTIT\_CTL.OS} = 0 \text{ AND } \text{CPL} = 0) \text{ OR} \\ & (\text{IA32\_RTIT\_CTL.USER} = 0 \text{ AND } \text{CPL} > 0) \text{ OR } (\text{IS\_IN\_A\_PRODUCTION\_ENCLAVE}^1) \text{ OR} \\ & (\text{IA32\_RTIT\_CTL.CR3Filter} = 1 \text{ AND } \text{IA32\_RTIT\_CR3\_MATCH} \text{ does not match CR3}) \end{aligned}$$

If the clearing of **ContextEn** causes **PacketEn** to be cleared, a Packet Generation Disable (`TIP.PGD`) packet is generated, but its IP payload is suppressed. If the setting of **ContextEn** causes **PacketEn** to be set, a Packet Generation Enable (`TIP.PGE`) packet is generated.

When **ContextEn** is 0, control flow packets (`TNT`, `FUP`, `TIP.*`, `MODE.*`) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as `MTC` and `PSB` (see Section 35.4.2.16 and Section 35.4.2.17), may still be generated while **ContextEn** is 0. For details of which packets are generated only when **ContextEn** is set, see Section 35.4.1.

The processor does not update **ContextEn** when **TriggerEn** = 0.

The value of **ContextEn** will toggle only when **TriggerEn** = 1.

1. Trace packets generation is disabled in a production enclave, see Section 35.2.8.5. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

### 35.2.5.4 Branch Enable (BranchEn)

This value is based purely on the IA32\_RTIT\_CTL.BranchEn value. If **BranchEn** is not set, then relevant COFI packets (TNT, TIP\*, FUP, MODE.\*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

### 35.2.5.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32\_RTIT\_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 35.2.4.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 35.4.1.

After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32\_RTIT\_CTL.ADDRn\_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT\_WRSTPRSV[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

## 35.2.6 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32\_RTIT\_CTL (see Section 35.2.7.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-cacheable (UC) stores. Software should avoid using MTRRs to mark any portion of the Intel PT output region as UC, as this may override the behavior described above and force Intel PT stores to UC, thereby incurring severe performance impact.

There is no guarantee that a packet will be written to memory or other trace endpoint after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated have reached their endpoint is to clear TraceEn and follow that with a store, fence, or serializing instruction; doing so ensures that all buffered packets are flushed out of the processor.

### 35.2.6.1 Single Range Output

When IA32\_RTIT\_CTL.ToPA and IA32\_RTIT\_CTL.FabricEn bits are clear, trace packet output is sent to a single, contiguous memory (or MMIO if DRAM is not available) range defined by a base address in IA32\_RTIT\_OUTPUT\_BASE (Section 35.2.7.7) and mask value in IA32\_RTIT\_OUTPUT\_MASK\_PTRS (Section 35.2.7.8). The current write pointer in this range is also stored in IA32\_RTIT\_OUTPUT\_MASK\_PTRS. This output range is circular, meaning that when the writes wrap around the end of the buffer they begin again at the base address.

This output method is best suited for cases where Intel PT output is either:

- Configured to be directed to a sufficiently large contiguous region of DRAM.
- Configured to go to an MMIO debug port, in order to route Intel PT output to a platform-specific trace endpoint (e.g., JTAG). In this scenario, a specific range of addresses is written in a circular manner, and SoC will intercept these writes and direct them to the proper device. Repeated writes to the same address do not overwrite each other, but are accumulated by the debugger, and hence no data is lost by the circular nature of the buffer.

The processor will determine the address to which to write the next trace packet output byte as follows:

```
OutputBase[63:0] ← IA32_RTIT_OUTPUT_BASE[63:0]
OutputMask[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[31:0])
OutputOffset[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[63:32])
trace_store_phys_addr ← (OutputBase & ~OutputMask) + (OutputOffset & OutputMask)
```

### Single-Range Output Errors

If the output base and mask are not properly configured by software, an operational error (see Section 35.3.9) will be signaled, and tracing disabled. Error scenarios with single-range output are:

- Mask value is non-contiguous.  
IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTablePointer value has a 0 in a less significant bit position than the most significant bit containing a 1.
- Base address and Mask are mis-aligned, and have overlapping bits set.  
IA32\_RTIT\_OUTPUT\_BASE && IA32\_RTIT\_OUTPUT\_MASK\_PTRS.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.

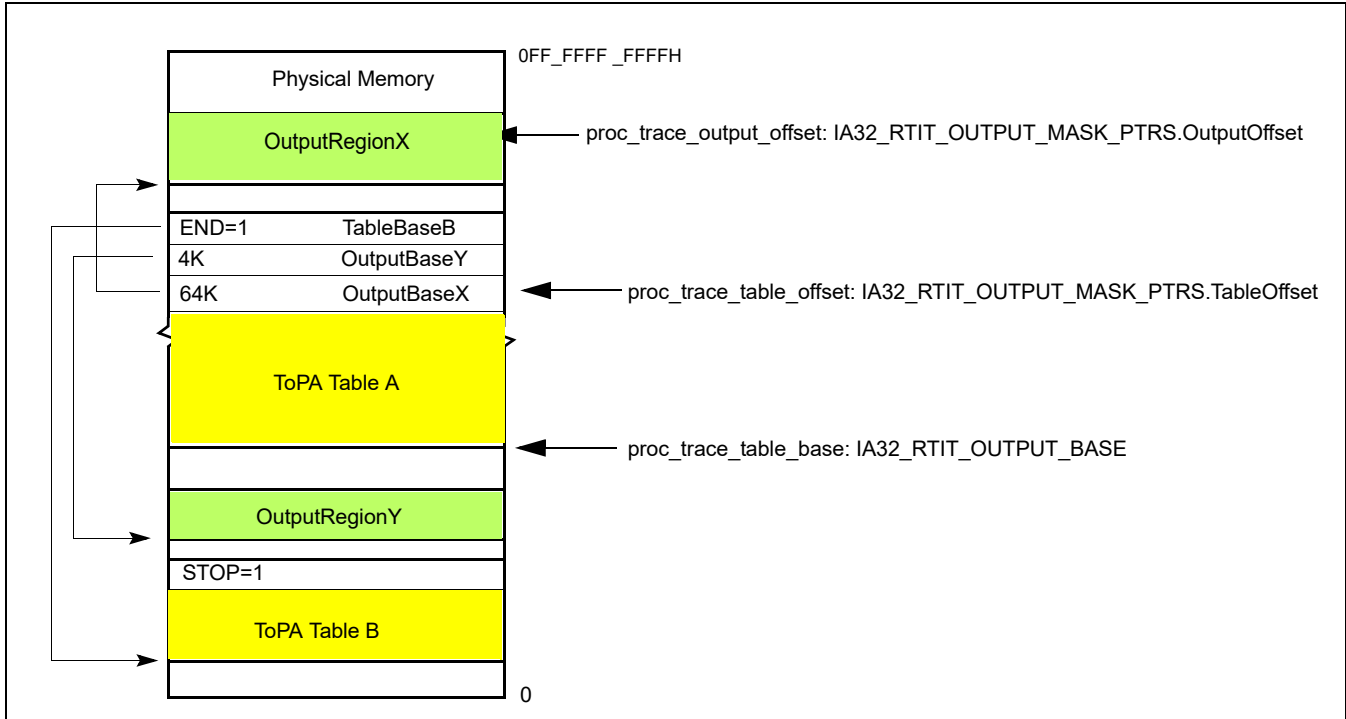
The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

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 is a pointer into the current output region and indicates the location of the next write. When tracing is

enabled, the processor loads this value from bits 63:32 (OutputOffset) of the IA32\_RTIT\_OUTPUT\_MASK\_PTRS. While tracing is enabled, the processor updates IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset with changes to proc\_trace\_output\_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc\_trace\_output\_offset.

Figure 35-1 provides an illustration (not to scale) of the table and associated pointers.



**Figure 35-1. ToPA Memory Illustration**

With the ToPA mechanism, the processor writes packets to the current output region (identified by `proc_trace_table_base` and the `proc_trace_table_offset`). The offset within that region to which the next byte will be written is identified by `proc_trace_output_offset`. When that region is filled with packet output (thus `proc_trace_output_offset = RegionSize-1`), `proc_trace_table_offset` is moved to the next ToPA entry, `proc_trace_output_offset` is set to 0, and packet writes begin filling the new output region specified by `proc_trace_table_offset`.

As packets are written out, each store derives its physical address as follows:

$$\text{trace\_store\_phys\_addr} \leftarrow \text{Base address from current ToPA table entry} + \text{proc\_trace\_output\_offset}$$

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Table 35-3 and the section that follows for details on STOP.

When an END entry is reached, the processor loads `proc_trace_table_base` with the base address held in this END entry, thereby moving the current table pointer to this new table. The `proc_trace_table_offset` is reset to 0, as is the `proc_trace_output_offset`, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (`proc_trace_table_offset = 01FFFFFFH`). In this case, the `proc_trace_table_offset` and `proc_trace_output_offset` are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32\_RTIT\_OUTPUT\_BASE and IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs while Intel PT is enabled may return stale values. Like all IA32\_RTIT\_\* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32\_RTIT\_CTL.TraceEn. This ensures that the output MSR values account for all packets generated to that point, after which the processor will cease updating the output MSR values until tracing resumes.<sup>1</sup>

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

### Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

### ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 35-2. The size of the address field is determined by the processor's physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

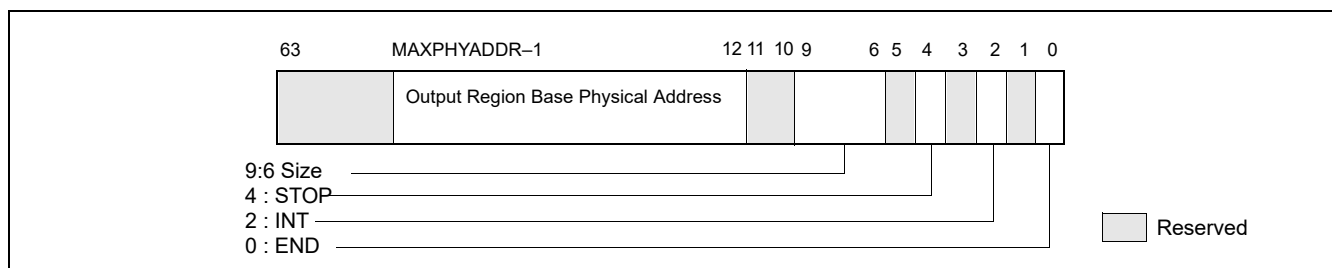


Figure 35-2. Layout of ToPA Table Entry

Table 35-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 35-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

Table 35-3. ToPA Table Entry Fields (Contd.)

ToPA Entry Field	Description
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.

### ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32\_RTIT\_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 35.2.7.4 for details on IA32\_RTIT\_STATUS.Stopped. This sequence is known as “ToPA Stop”.

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32\_RTIT\_OUTPUT\_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOffsetTableOffset 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.

When IA32\_RTIT\_CTL.InjectPsbPmiOnEnable[56] = 1, the PMI handler should take the following actions:

1. Ignore ToPA PMIs that are taken when TraceEn = 0, because the Intel PT MSR state may have already been saved by XSAVES, and because the PMI will be re-injected when Intel PT is re-enabled.
2. Clear the new IA32\_RTIT\_STATUS.PendTopaPMI[7] bit once the PMI has been handled. This bit should not be cleared in cases where a PMI is ignored due to TraceEn = 0.

### 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. On processors that support PMI preservation (CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1), setting IA32\_RTIT\_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PMI that is pending at the time PT is disabled will be recorded by setting IA32\_RTIT\_STATUS.PendTopaPMI[7] = 1. A PMI will then be pended when the saved PT context is later restored.

Table 35-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

**Table 35-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS**

Pseudo Code Flow
<pre> IF (IA32_PERF_GLOBAL_STATUS.ToPA)   Save IA32_RTIT_CTL value;   IF ( IA32_RTIT_CTL.TraceEN )     Disable Intel PT by clearing TraceEn;   FI;   IF ( there is space available to grow the current ToPA table )     Add one or more ToPA entries after the last entry in the ToPA table;     Point new ToPA entry address field(s) to new output region base(s);   ELSE     Modify an upcoming ToPA entry in the current table to have END=1;     IF (output should transition to a new ToPA table )       Point the address of the "END=1" entry of the current table to the new table base;     ELSE       /* Continue to use the current ToPA table, make a circular. */       Point the address of the "END=1" entry to the base of the current table;       Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;       /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */     FI;   FI;   Restore saved IA32_RTIT_CTL.value; FI; </pre>

### ToPA Errors

When a malformed ToPA entry is found, an operational error results (see Section 35.3.9). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation.**  
This describes cases where a bit marked as reserved in Section 35.2.6.2 above is set to 1.
2. **ToPA alignment violation.**  
This includes cases where illegal ToPA entry base address bits are set to 1:
  - a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32\_RTIT\_OUTPUT\_BASE, or from a ToPA entry with END=1.
  - b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
  - 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 write pointer ( <code>IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset &lt;&lt; 3)</code> ) 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 35.2.5.4 for details on BranchEn.
17:14	MTCFreq	0	Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined: 0: ART(0), 1: ART(1), 2: ART(2), 3: ART(3), 4: ART(4), 5: ART(5), 6: ART(6), 7: ART(7), 8: ART(8), 9: ART(9), 10: ART(10), 11: ART(11), 12: ART(12), 13: ART(13), 14: ART(14), 15: ART(15) Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
18	Reserved	0	Must be 0.
22:19	CycThresh	0	CYC packet threshold, see Section 35.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as $2^{(CycThresh-1)}$ . The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384 Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
23	Reserved	0	Must be 0.

Table 35-6. IA32\_RTIT\_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
27:24	PSBFreq	0	Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
31:28	Reserved	0	Must be 0.
35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGE CNT[2:0] >= 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.RANGE CNT[2:0] < 2.
43:40	ADDR2_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGE CNT[2:0] < 3.

Table 35-6. IA32\_RTIT\_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
47:44	ADDR3_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECONT[2:0] < 4.
55:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
56	InjectPsbPmiOnEnable	0	1: Enables use of IA32_RTIT_STATUS bits PendPSB[6] and PendTopaPMI[7], see Section 35.2.7.4, "IA32_RTIT_STATUS MSR" for behavior of these bits. 0: IA32_RTIT_STATUS bits 6 and 7 are ignored. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 0.
59:57	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
63:60	Reserved	0	Must be 0.

### 35.2.7.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 35.4.2.17) will be generated if IA32\_RTIT\_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 35.4.2).

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.
6	PendPSB	0	If IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, the processor sets this bit when the threshold for a PSB+ to be inserted has been reached. The processor will clear this bit when the PSB+ has been inserted into the trace. If PendPSB = 1 and InjectPsbPmiOnEnable = 1 when IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a PSB+ will be inserted into the trace.  This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1.
7	PendTopaPMI	0	If IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, the processor sets this bit when the threshold for a ToPA PMI to be inserted has been reached. Software should clear this bit once the ToPA PMI has been handled, see "ToPA PMI" for details. If PendTopaPMI = 1 and InjectPsbPmiOnEnable = 1 when IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a PMI will be pended.  This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1.
31:8	Reserved	0	Must be 0.
48:32	PacketByteCnt	0	This field is written by the processor, and holds a count of packet bytes that have been sent out. The processor also uses this field to determine when the next PSB packet should be inserted. Note that the processor may clear or modify this field at any time while IA32_RTIT_CTL.TraceEn=1. It will have a stable value when IA32_RTIT_CTL.TraceEn=0. See Section 35.4.2.17 for details.
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	<p>The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value.</p> <p>The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 35.2.7.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 35.3.9).</p> <p>1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See “ToPA Errors” in Section 35.2.6.2 as well as Section 35.3.9.</p>
63:MAXPHYADDR	Reserved	0	Must be 0.

**35.2.7.8 IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR**

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 35.2.6.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32\_RTIT\_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32\_RTIT\_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

**Table 35-9. IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR**

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. The size of this field indicates that regions can be of size 128B up to 4GB. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 35.3.9) will be signaled when TraceEn is set.</p> <p>1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.</p>



Table 35-9. IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOffsetTableOffset field, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.</p> <p>1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.</p>

## 35.2.8 Interaction of Intel® Processor Trace and Other Processor Features

### 35.2.8.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 35-10 for details.

Table 35-10. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP)
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: <ul style="list-style-type: none"> <li>▪ Nested XBEGIN or XACQUIRE lock</li> <li>▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set)</li> <li>▪ Non-outermost XEND or XRELEASE lock</li> </ul>	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 (Intel® SGX)

Intel SGX provides an application with the ability to instantiate a protective container (an enclave) with confidentiality and integrity (see the *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and Intel SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. An enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.

During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 35.7.

### Debug Enclaves

Intel SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see the *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by an enclave entry or exit. Hence, the generation of ContextEn-dependent packets within a debug enclave is allowed.

#### 35.2.8.6 SENTER/ENTERACCS and ACM

GETSEC[SENDER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

#### 35.2.8.7 Intel® Memory Protection Extensions (Intel® MPX)

Bounds exceptions (#BR) caused by Intel MPX are treated like other exceptions, producing FUP and TIP packets that indicate the source and destination IPs.

## 35.3 CONFIGURATION AND PROGRAMMING GUIDELINE

### 35.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 35-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.

**Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities**

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
EAX	31:0	Maximum valid sub-leaf Index	Specifies the index of the maximum valid sub-leaf for this CPUID leaf
EBX	0	CR3 Filtering Support	1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. See Section 35.2.7. 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CR3_MATCH, will #GP fault.
	1	Configurable PSB and Cycle-Accurate Mode Supported	1: (a) IA32_RTIT_CTL.PSBFreq can be set to a non-zero value, in order to select the preferred PSB frequency (see below for allowed values). (b) IA32_RTIT_STATUS.PacketByteCnt can be set to a non-zero value, and will be incremented by the processor when tracing to indicate progress towards the next PSB. If trace packet generation is enabled by setting TraceEn, a PSB will only be generated if PacketByteCnt=0. (c) IA32_RTIT_CTL.CYCEn can be set to 1 to enable Cycle-Accurate Mode. See Section 35.2.7. 0: (a) Any attempt to set IA32_RTIT_CTL.PSBFreq, to set IA32_RTIT_CTL.CYCEn, or write a non-zero value to IA32_RTIT_STATUS.PacketByteCnt any access to IA32_RTIT_CR3_MATCH, will #GP fault. (b) If trace packet generation is enabled by setting TraceEn, a PSB is always generated. (c) Any attempt to set IA32_RTIT_CTL.CYCEn will #GP fault.
	2	IP Filtering and TraceStop supported, and Preserve Intel PT MSRs across warm reset	1: (a) IA32_RTIT_CTL provides at one or more ADDRn_CFG field to configure the corresponding address range MSRs for IP Filtering or IP TraceStop. Each ADDRn_CFG field accepts a value in the range of 0:2 inclusive. The number of ADDRn_CFG fields is reported by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0]. (b) At least one register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B are provided to configure address ranges for IP filtering or IP TraceStop. (c) On warm reset, all Intel PT MSRs will retain their pre-reset values, though IA32_RTIT_CTL.TraceEn will be cleared. The Intel PT MSRs are listed in Section 35.2.7. 0: (a) An Attempt to write IA32_RTIT_CTL.ADDRn_CFG with non-zero encoding values will cause #GP. (b) Any access to IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B, will #GP fault. (c) On warm reset, all Intel PT MSRs will be cleared.
	3	MTC Supported	1: IA32_RTIT_CTL.MTCEn can be set to 1, and MTC packets will be generated. See Section 35.2.7. 0: An attempt to set IA32_RTIT_CTL.MTCEn or IA32_RTIT_CTL.MTCFreq to a non-zero value will #GP fault.
	4	PTWRITE Supported	1: Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. 0: Writes that set IA32_RTIT_CTL[12] or IA32_RTIT_CTL[5] will #GP, and PTWRITE will #UD fault.
	5	Power Event Trace Supported	1: Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. 0: Writes that set IA32_RTIT_CTL[4] will #GP.
		31:6	Reserved

Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities (Contd.)

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
ECX	0	ToPA Output Supported	1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 35.2.6.2) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Unless CPUID.(EAX=14H, ECX=0);ECX.SNGLRNGOUT[bit 2] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	1	ToPA Tables Allow Multiple Output Entries	1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOffsetTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 35.2.6.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see “ToPA Errors” in Section 35.2.6.2).
	2	Single-Range Output Supported	1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0);ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	3	Output to Trace Transport Subsystem Supported	1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will #GP fault.
	30:4	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.

Table 35-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=1)		Name	Description Behavior
Register	Bits		
EAX	2:0	Number of Address Ranges	A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. <b>NOTE:</b> 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_CTL.MTCFreq 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.<sup>1</sup>

**Table 35-13. Memory Layout of the Trace Configuration State Component**

Offset within Component Area	Field	Offset within Component Area	Field
0H	IA32_RTIT_CTL	08H	IA32_RTIT_OUTPUT_BASE
10H	IA32_RTIT_OUTPUT_MASK_PTRS	18H	IA32_RTIT_STATUS
20H	IA32_RTIT_CR3_MATCH	28H	IA32_RTIT_ADDR0_A
30H	IA32_RTIT_ADDR0_B	38H	IA32_RTIT_ADDR1_A
40H	IA32_RTIT_ADDR1_B	48H-End	Reserved

The IA32\_XSS MSR is zero coming out of RESET. Once IA32\_XSS[bit 8] is set, system software operating at CPL=0 can use XSAVES/XRSTORS with the appropriate requested-feature bitmap (RFBM) to manage supervisor state components in the XSAVE map. See Chapter 13, “Managing State Using the XSAVE Feature Set” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### 35.3.6 Cycle-Accurate Mode

Intel PT can be run in a cycle-accurate mode which enables CYC packets (see Section 35.4.2.14) that provide low-level information in the processor core clock domain. This cycle counter data in CYC packets can be used to compute IPC (Instructions Per Cycle), or to track wall-clock time on a fine-grain level.

To enable cycle-accurate mode packet generation, software should set IA32\_RTIT\_CTL.CYCEn=1. It is recommended that software also set TSCEn=1 anytime cycle-accurate mode is in use. With this, all CYC-eligible packets will be preceded by a CYC packet, the payload of which indicates the number of core clock cycles since the last CYC packet. In cases where multiple CYC-eligible packets are generated in a single cycle, only a single CYC will be generated before the CYC-eligible packets, otherwise each CYC-eligible packet will be preceded by its own CYC. The CYC-eligible packets are:

- TNT, TIP, TIP.PGE, TIP.PGD, MODE.EXEC, MODE.TSX, PIP, VMCS, OVF, MTC, TSC, PTWRITE, EXSTOP

TSC packets are generated when there is insufficient information to reconstruct wall-clock time, due to tracing being disabled (TriggerEn=0), or power down scenarios like a transition to a deep-sleep MWAIT C-state. In this case, the CYC that is generated along with the TSC will indicate the number of cycles actively tracing (those powered up, with TriggerEn=1) executed between the last CYC packet and the TSC packet. And hence the amount of time spent while tracing is inactive can be inferred from the difference in time between that expected based on the CYC value, and the actual time indicated by the TSC.

Additional CYC packets may be sent stand-alone, so that the processor can ensure that the decoder is aware of the number of cycles that have passed before the internal hardware counter wraps, or is reset due to other micro-architectural condition. There is no guarantee at what intervals these standalone CYC packets will be sent, except that they will be sent before the wrap occurs. An illustration is given below.

1. Table 35-13 documents support for the MSRs defining address ranges 0 and 1. Processors that provide XSAVE support for Intel Processor Trace support only those address ranges.

**Example 35-1. An Illustrative CYC Packet Example**

Time (cycles)	Instruction Snapshot	Generated Packets	Comment
x	call %eax	CYC(?), TIP	?Elapsed cycles from the previous CYC unknown
x + 2	call %ebx	CYC(2), TIP	1 byte CYC packet; 2 cycles elapsed from the previous CYC
x + 8	jnz Foo (not taken)	CYC(6)	1 byte CYC packet
x + 9	ret (compressed)		
x + 12	jnz Bar (taken)		
x + 16	ret (uncompressed)	TNT, CYC(8), TIP	1 byte CYC packet
x + 4111		CYC(4095)	2 byte CYC packet
x + 12305		CYC(8194)	3 byte CYC packet
x + 16332	mov cr3, %ebx	CYC(4027), PIP	2 byte CYC packet

**35.3.6.1 Cycle Counter**

The cycle counter is implemented in hardware (independent of the time stamp counter or performance monitoring counters), and is a simple incrementing counter that does not saturate, but rather wraps. The size of the counter is implementation specific.

The cycle counter is reset to zero any time that TriggerEn is cleared, and when a CYC packet is sent. The cycle counter will continue to count when ContextEn or FilterEn are cleared, and cycle packets will still be generated. It will not count during sleep states that result in Intel PT logic being powered-down, but will count up to the point where clocks are disabled, and resume counting once they are re-enabled.

**35.3.6.2 Cycle Packet Semantics**

Cycle-accurate mode adheres to the following protocol:

- All packets that precede a CYC packet represent instructions or events that took place before the CYC time.
- All packets that follow a CYC packet represent instructions or events that took place at the same time as, or after, the CYC time.
- The CYC-eligible packet that immediately follows a CYC packet represents an instruction or event that took place at the same time as the CYC time.

These items above give the decoder a means to apply CYC packets to a specific instruction in the assembly stream. Most packets represent a single instruction or event, and hence the CYC packet that precedes each of those packets represents the retirement time of that instruction or event. In the case of TNT packets, up to 6 conditional branches and/or compressed RETs may be contained in the packet. In this case, the preceding CYC packet provides the retirement time of the first branch in the packet. It is possible that multiple branches retired in the same cycle as that first branch in the TNT, but the protocol will not make that obvious. Also note that a MTC packet could be generated in the same cycle as the first JCC in the TNT packet. In this case, the CYC would precede both the MTC and the TNT, and apply to both.

Note that there are times when the cycle counter will stop counting, though cycle-accurate mode is enabled. After any such scenario, a CYC packet followed by TSC packet will be sent. See Section 35.8.3.2 to understand how to interpret the payload values

**Multi-packet Instructions or Events**

Some operations, such as interrupts or task switches, generate multiple packets. In these cases, multiple CYC packets may be sent for the operation, preceding each CYC-eligible packet in the operation. An example, using a task switch on a software interrupt, is shown below.

**Example 35-2. An Example of CYC in the Presence of Multi-Packet Operations**

Time (cycles)	Instruction Snapshot	Generated Packets
x	jnz Foo (not taken)	CYC(?),
x + 2	ret (compressed)	
x + 8	jnz Bar (taken)	
x + 9	jmp %eax	TNT, CYC(9), TIP
x + 12	jnz Bar (not taken)	CYC(3)
x + 32	int3 (task gate)	TNT, FUP, CYC(10), PIP, CYC(20), MODE.Exec, TIP

**35.3.6.3 Cycle Thresholds**

Software can opt to reduce the frequency of cycle packets, a trade-off to save bandwidth and intrusion at the expense of precision. This is done by utilizing a cycle threshold (see Section 35.2.7.2).

IA32\_RTIT\_CTL.CycThresh indicates to the processor the minimum number of cycles that must pass before the next CYC packet should be sent. If this value is 0, no threshold is used, and CYC packets can be sent every cycle in which a CYC-eligible packet is generated. If this value is greater than 0, the hardware will wait until the associated number of cycles have passed since the last CYC packet before sending another. CPUID provides the threshold options for CycThresh, see Section 35.3.1.

Note that the cycle threshold does not dictate how frequently a CYC packet will be posted, it merely assigns the maximum frequency. If the cycle threshold is 16, a CYC packet can be posted no more frequently than every 16 cycles. However, once that threshold of 16 cycles has passed, it still requires a new CYC-eligible packet to be generated before a CYC will be inserted. Table 35-14 illustrates the threshold behavior.

**Table 35-14. An Illustrative CYC Packet Example**

Time (cycles)	Instruction Snapshot	Threshold			
		0	16	32	64
x	jmp %eax	CYC, TIP	CYC, TIP	CYC, TIP	CYC, TIP
x + 9	call %ebx	CYC, TIP	TIP	TIP	TIP
x + 15	call %ecx	CYC, TIP	TIP	TIP	TIP
x + 30	jmp %edx	CYC, TIP	CYC, TIP	TIP	TIP
x + 38	mov cr3, %eax	CYC, PIP	PIP	CYC, PIP	PIP
x + 46	jmp [%eax]	CYC, TIP	CYC, TIP	TIP	TIP
x + 64	call %edx	CYC, TIP	CYC, TIP	TIP	CYC, TIP
x + 71	jmp %edx	CYC, TIP	TIP	CYC, TIP	TIP

**35.3.7 Decoder Synchronization (PSB+)**

The PSB packet (Section 35.4.2.17) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well. The decoder should never need to retain any information (e.g., LastIP, call stack, compound packet event) across a PSB; all compound packet events will be completed before a PSB, and any compression state will be reset.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 35.4.2.18). One or more packets may be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as “status only”, since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the

normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:

- Timestamp (TSC), if IA32\_RTIT\_CTL.TSCEn=1.
- Timestamp-MTC Align (TMA), if IA32\_RTIT\_CTL.TSCEn=1 && IA32\_RTIT\_CTL.MTCEn=1.
- Paging Information Packet (PIP), if ContextEn=1 and IA32\_RTIT\_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the “conceal VMX from PT” VM-execution control is 0.
- VMCS packet, if either the logical is in VMX root operation or the logical processor is in VMX non-root operation and the “conceal VMX from PT” VM-execution control is 0.
- Core Bus Ratio (CBR).
- MODE.TSX, if ContextEn=1 and BranchEn = 1.
- MODE.Exec, if PacketEn=1.
- Flow Update Packet (FUP), if PacketEn=1.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies. The ordering of packets within PSB+ is not fixed. Timing packets such as CYC and MTC may be generated between PSB and PSBEND, and their meanings are the same as outside PSB+.

A PSB+ can be lost in some scenarios. If IA32\_RTIT\_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32\_RTIT\_CTL.TraceEn, a VM-exit that clears IA32\_RTIT\_CTL.TraceEn, an #SMI, or any time that either IA32\_RTIT\_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32\_RTIT\_STATUS.Error is set (e.g., by an Intel PT output error).

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+. If IA32\_RTIT\_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32\_RTIT\_CTL.TraceEn, a VM-exit that clears IA32\_RTIT\_CTL.TraceEn, an #SMI, or any time that either IA32\_RTIT\_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32\_RTIT\_STATUS.Error is set (e.g., by an Intel PT output error). On processors that support PSB preservation (CPUID.(EAX=14H, ECX=0):EBX.INJECTPSBPMI[6] = 1), setting IA32\_RTIT\_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PSB+ that is pending at the time PT is disabled will be recorded by setting IA32\_RTIT\_STATUS.PendPSB[6] = 1. A PSB will then be pended when the saved PT context is later restored.

### 35.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor’s dedicated internal buffers are all full, an “internal buffer overflow” occurs. On such an overflow packet generation ceases (as packets would need to enter the processor’s internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 35.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32\_RTIT\_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. If the overflow resolves while PacketEn=1, only timing packets may come between the OVF and the FUP. If the overflow resolves while PacketEn=0, any other packets that are not dependent on PacketEn may come between the OVF and the TIP.PGE.

#### 35.3.8.1 Overflow Impact on Enables

The address comparisons to ADDRn ranges, for IP filtering and TraceStop (Section 35.2.4.3), continue during a buffer overflow, and TriggerEn, ContextEn, and FilterEn may change during a buffer overflow. Like other packets, however, any TIP.PGE or TIP.PGD packets that would have been generated will be lost. Further, IA32\_RTIT\_STATUS.PacketByteCnt will not increment, since it is only incremented when packets are generated.

If a TraceStop event occurs during the buffer overflow, IA32\_RTIT\_STATUS.Stopped will still be set, tracing will cease as a result. However, the TraceStop packet, and any TIP.PGD that result from the TraceStop, may be dropped.

### 35.3.8.2 Overflow Impact on Timing Packets

Any timing packets that are generated during a buffer overflow will be dropped. If only a few MTC packets are dropped, a decoder should be able to detect this by noticing that the time value in the first MTC packet after the buffer overflow incremented by more than one. If the buffer overflow lasted long enough that 256 MTC packets are lost (and thus the MTC packet 'wraps' its 8-bit CTC value), then the decoder may be unable to properly understand the trace. This is not an expected scenario. No CYC packets are generated during overflow, even if the cycle counter wraps.

Note that, if cycle-accurate mode is enabled, the OVF packet will generate a CYC packet. Because the cycle counter counts during overflows, this CYC packet can provide the duration of the overflow. However, there is a risk that the cycle counter wrapped during the overflow, which could render this CYC misleading.

### 35.3.9 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See "ToPA Errors" in Section 35.2.6.2 for details on ToPA errors, and Section 35.2.6.4 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32\_RTIT\_STATUS.Error is set, which will cause IA32\_RTIT\_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32\_RTIT\_STATUS.Error. At this point, packet generation can be re-enabled.

## 35.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

### 35.4.1 Packet Relationships and Ordering

This section introduces the concept of packet "binding", which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 35.4.2 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

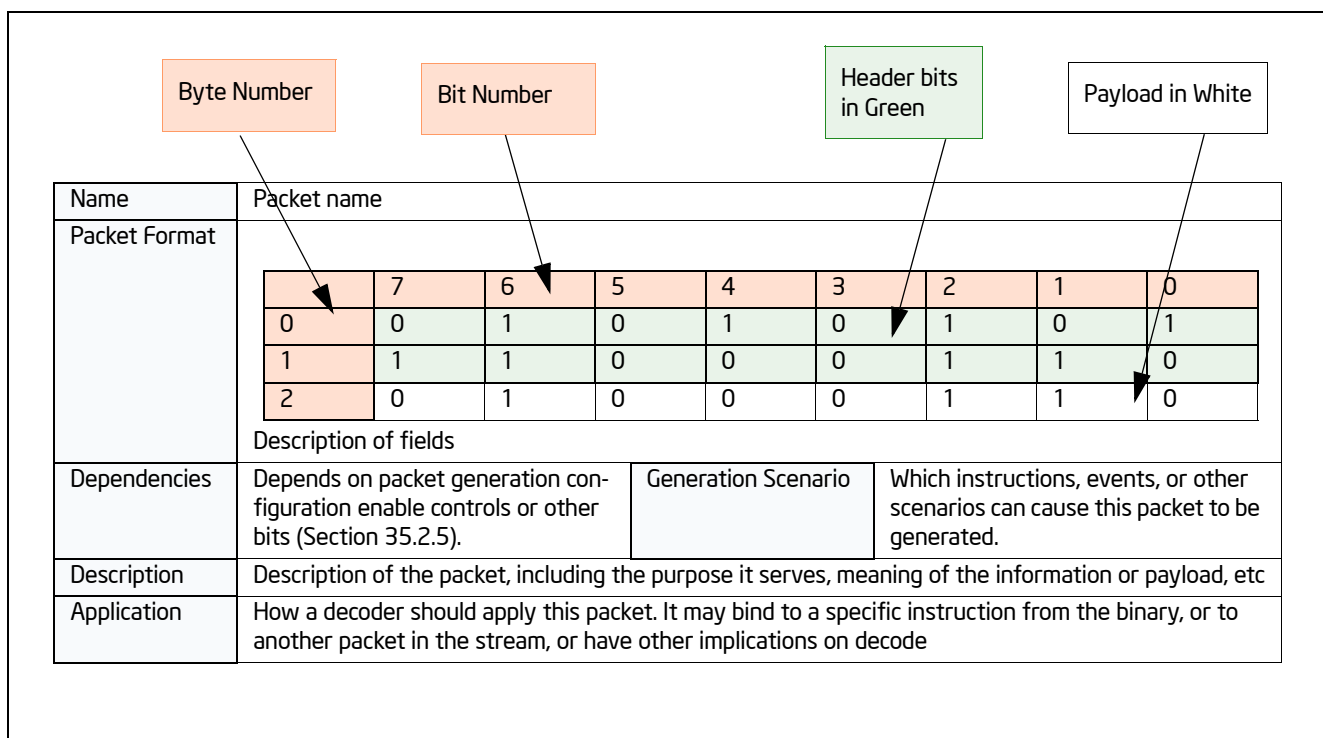
indicated by these intermediate packets should be applied at the IP of the TIP\* packet. A summary of compound packet events is provided in Table 35-15; see Section 35.4.2 for more per-packet details and Section 35.7 for more detailed packet generation examples.

**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 <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	0	Short TNT
	B <sub>1</sub> ...B <sub>N</sub> represent the last N conditional branch or compressed RET (Section 35.4.2.2) results, such that B <sub>1</sub> is oldest and B <sub>N</sub> 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.									
		7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	1	0	Long TNT
	1	1	0	1	0	0	0	1	1	
	2	B <sub>40</sub>	B <sub>41</sub>	B <sub>42</sub>	B <sub>43</sub>	B <sub>44</sub>	B <sub>45</sub>	B <sub>46</sub>	B <sub>47</sub>	
	3	B <sub>32</sub>	B <sub>33</sub>	B <sub>34</sub>	B <sub>35</sub>	B <sub>36</sub>	B <sub>37</sub>	B <sub>38</sub>	B <sub>39</sub>	
	4	B <sub>24</sub>	B <sub>25</sub>	B <sub>26</sub>	B <sub>27</sub>	B <sub>28</sub>	B <sub>29</sub>	B <sub>30</sub>	B <sub>31</sub>	
	5	B <sub>16</sub>	B <sub>17</sub>	B <sub>18</sub>	B <sub>19</sub>	B <sub>20</sub>	B <sub>21</sub>	B <sub>22</sub>	B <sub>23</sub>	
	6	B <sub>8</sub>	B <sub>9</sub>	B <sub>10</sub>	B <sub>11</sub>	B <sub>12</sub>	B <sub>13</sub>	B <sub>14</sub>	B <sub>15</sub>	
	7	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	B <sub>7</sub>	
	Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these "partial TNTs" are shown below.									
		7	6	5	4	3	2	1	0	
	0	0	0	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	0	Short TNT
		7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	1	0	Long TNT
	1	1	0	1	0	0	0	1	1	
	2	B <sub>24</sub>	B <sub>25</sub>	B <sub>26</sub>	B <sub>27</sub>	B <sub>28</sub>	B <sub>29</sub>	B <sub>30</sub>	B <sub>31</sub>	
	3	B <sub>16</sub>	B <sub>17</sub>	B <sub>18</sub>	B <sub>19</sub>	B <sub>20</sub>	B <sub>21</sub>	B <sub>22</sub>	B <sub>23</sub>	
	4	B <sub>8</sub>	B <sub>9</sub>	B <sub>10</sub>	B <sub>11</sub>	B <sub>12</sub>	B <sub>13</sub>	B <sub>14</sub>	B <sub>15</sub>	
	5	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	B <sub>7</sub>	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
Dependencies	PacketEn			Generation Scenario		On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated, or certain micro-architectural conditions occurring, before the TNT is full.				

**Table 35-16. TNT Packet Definition (Contd.)**

Description	Provides the taken/not-taken results for the last 1-N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 35.4.2.2). The TNT payload bits should be interpreted as follows: <ul style="list-style-type: none"> <li>▪ 1 indicates a taken conditional branch, or a compressed RET</li> <li>▪ 0 indicates a not-taken conditional branch</li> </ul>
Application	Each valid payload bit (that is, bits between the header bits and the trailing Stop bit) applies to an upcoming conditional branch or RET instruction. Once a decoder consumes a TNT packet with N valid payload bits, these bits should be applied to (and hence provide the destination for) the next N conditional branches or RETs

### 35.4.2.2 Target IP (TIP) Packet

**Table 35-17. IP Packet Definition**

Name	Target IP (TIP) Packet																																																																																												
Packet Format	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 10%;"></th> <th style="width: 10%;">7</th> <th style="width: 10%;">6</th> <th style="width: 10%;">5</th> <th style="width: 10%;">4</th> <th style="width: 10%;">3</th> <th style="width: 10%;">2</th> <th style="width: 10%;">1</th> <th style="width: 10%;">0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td colspan="3">IPBytes</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">TargetIP[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">TargetIP[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">TargetIP[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">TargetIP[31:24]</td> </tr> <tr> <td>5</td> <td colspan="8">TargetIP[39:32]</td> </tr> <tr> <td>6</td> <td colspan="8">TargetIP[47:40]</td> </tr> <tr> <td>7</td> <td colspan="8">TargetIP[55:48]</td> </tr> <tr> <td>8</td> <td colspan="8">TargetIP[63:56]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	IPBytes			0	1	1	0	1	1	TargetIP[7:0]								2	TargetIP[15:8]								3	TargetIP[23:16]								4	TargetIP[31:24]								5	TargetIP[39:32]								6	TargetIP[47:40]								7	TargetIP[55:48]								8	TargetIP[63:56]							
	7	6	5	4	3	2	1	0																																																																																					
0	IPBytes			0	1	1	0	1																																																																																					
1	TargetIP[7:0]																																																																																												
2	TargetIP[15:8]																																																																																												
3	TargetIP[23:16]																																																																																												
4	TargetIP[31:24]																																																																																												
5	TargetIP[39:32]																																																																																												
6	TargetIP[47:40]																																																																																												
7	TargetIP[55:48]																																																																																												
8	TargetIP[63:56]																																																																																												
Dependencies	PacketEn	Generation Scenario	Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX <sup>1</sup> .																																																																																										
Description	Provides the target for some control flow transfers																																																																																												
Application	<p>Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</p> <p>The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</p>																																																																																												

**NOTES:**

1. EENTER, EEXIT, ERESUME, AEX would be possible only for a debug enclave.

### IP Compression

The IP payload in a TIP, FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the "Last IP" that was encoded in trace packets, thus the decoder will need to keep track of the "Last IP" state in software, to match fidelity with packets generated by hardware. "Last IP" is initialized to zero, hence if the first IP in the trace may be compressed if the upper bytes are zeroes.



The “IPBytes” field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

**Table 35-18. 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 ... // <b>an asynchronous interrupt arrives</b> INTHandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)

### 35.4.2.4 Packet Generation Enable (TIP.PGE) Packet

**Table 35-20. TIP.PGE Packet Definition**

Name	Target IP - Packet Generation Enable (TIP.PGE) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			1	0	0	0	1
	1	TargetIP[7:0]							
	2	TargetIP[15:8]							
	3	TargetIP[23:16]							
	4	TargetIP[31:24]							
	5	TargetIP[39:32]							
	6	TargetIP[47:40]							
	7	TargetIP[55:48]							
	8	TargetIP[63:56]							
Dependencies	PacketEn transitions to 1			Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn				
Description	<p>Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins. This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:</p> <ul style="list-style-type: none"> <li>▪ 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.</li> <li>▪ 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.</li> <li>▪ 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.</li> </ul>								
Application	TIP.PGE packets bind to the instruction at the IP given in the payload.								

### 35.4.2.5 Packet Generation Disable (TIP.PGD) Packet

**Table 35-21. TIP.PGD Packet Definition**

Name	Target IP - Packet Generation Disable (TIP.PGD) Packet																																																																																																	
Packet Format	<table border="1" data-bbox="318 380 1305 751"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td colspan="3">IPBytes</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">TargetIP[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">TargetIP[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">TargetIP[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">TargetIP[31:24]</td> </tr> <tr> <td>5</td> <td colspan="8">TargetIP[39:32]</td> </tr> <tr> <td>6</td> <td colspan="8">TargetIP[47:40]</td> </tr> <tr> <td>7</td> <td colspan="8">TargetIP[55:48]</td> </tr> <tr> <td>8</td> <td colspan="8">TargetIP[63:56]</td> </tr> </table>									7	6	5	4	3	2	1	0	0	IPBytes			0	0	0	0	1	1	TargetIP[7:0]								2	TargetIP[15:8]								3	TargetIP[23:16]								4	TargetIP[31:24]								5	TargetIP[39:32]								6	TargetIP[47:40]								7	TargetIP[55:48]								8	TargetIP[63:56]							
	7	6	5	4	3	2	1	0																																																																																										
0	IPBytes			0	0	0	0	1																																																																																										
1	TargetIP[7:0]																																																																																																	
2	TargetIP[15:8]																																																																																																	
3	TargetIP[23:16]																																																																																																	
4	TargetIP[31:24]																																																																																																	
5	TargetIP[39:32]																																																																																																	
6	TargetIP[47:40]																																																																																																	
7	TargetIP[55:48]																																																																																																	
8	TargetIP[63:56]																																																																																																	
Dependencies	PacketEn transitions to 0	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn																																																																																															
Description	<p>Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn = 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn.</p> <p>PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:</p> <ul style="list-style-type: none"> <li>▪ 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.</li> <li>▪ FilterEn: This is cleared when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes = 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.</li> <li>▪ 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.</li> </ul> <p>Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNT bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (ie, SYSCALL when IA32_RTIT_CTL.OS=0, In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.</p>																																																																																																	
Application	<p>TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce.</p> <p>In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.</p> <p>If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.</p>																																																																																																	

### 35.4.2.6 Flow Update (FUP) Packet

Table 35-22. FUP Packet Definition

Name	Flow Update (FUP) Packet																																																																																												
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td colspan="3">IPBytes</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">IP[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">IP[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">IP[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">IP[31:24]</td> </tr> <tr> <td>5</td> <td colspan="8">IP[39:32]</td> </tr> <tr> <td>6</td> <td colspan="8">IP[47:40]</td> </tr> <tr> <td>7</td> <td colspan="8">IP[55:48]</td> </tr> <tr> <td>8</td> <td colspan="8">IP[63:56]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	IPBytes			1	1	1	0	1	1	IP[7:0]								2	IP[15:8]								3	IP[23:16]								4	IP[31:24]								5	IP[39:32]								6	IP[47:40]								7	IP[55:48]								8	IP[63:56]							
	7	6	5	4	3	2	1	0																																																																																					
0	IPBytes			1	1	1	0	1																																																																																					
1	IP[7:0]																																																																																												
2	IP[15:8]																																																																																												
3	IP[23:16]																																																																																												
4	IP[31:24]																																																																																												
5	IP[39:32]																																																																																												
6	IP[47:40]																																																																																												
7	IP[55:48]																																																																																												
8	IP[63:56]																																																																																												
Dependencies	TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 35.2.4 for details.)	Generation Scenario	Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit, #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, EENTER, EEXIT, ERESUME, EEE, AEX, <sup>1</sup> INTO, INT1, INT3, INT <i>n</i> , a WRMSR that disables packet generation.																																																																																										
Description	Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.																																																																																												
Application	<p>FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.</p> <p>In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 35.4.2.8 for details.</p> <p>Otherwise, FUPs are part of compound packet events (see Section 35.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) packet will provide the destination IP. Other packets may be included in the compound event between the FUP and TIP.</p>																																																																																												

**NOTES:**

1. EENTER, EEXIT, ERESUME, EEE, AEX apply only if Intel Software Guard Extensions is supported.

### FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 35-23 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 <sup>1</sup>	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

Name	Paging Information (PIP) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	1	0
	1	0	1	0	0	0	0	1	1
	2	CR3[11:5] or 0							RSVD/NR
	3	CR3[19:12]							
	4	CR3[27:20]							
	5	CR3[35:28]							
	6	CR3[43:36]							
	7	CR3[51:44]							
Dependencies	TriggerEn && ContextEn && IA32_RTIT_CTL.OS			Generation Scenario	MOV CR3, Task switch, INIT, SIPI, PSB+, VM exit, VM entry				
Description	<p>The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other paging modes (32-bit and 4-level paging<sup>1</sup>), these bits are 0.</p> <p>This packet holds the CR3 address value. It will be generated on operations that modify CR3:</p> <ul style="list-style-type: none"> <li>▪ MOV CR3 operation</li> <li>▪ Task Switch</li> <li>▪ INIT and SIPI</li> <li>▪ VM exit, if “conceal VMX from PT” VM-exit control is 0 (see Section 35.5.1)</li> <li>▪ VM entry, if “conceal VMX from PT” VM-entry control is 0</li> </ul> <p>PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section 35.2.8.3 for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.</p> <p>The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.</p> <p>The PIP packet contains the new CR3 value when CR3 is written.</p> <p>PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX from PT” VM-execution control is 0 (see Section 35.5.1). All other PIPs clear the NR bit.</p>								
Application	<p>The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:</p> <ol style="list-style-type: none"> <li>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.</li> <li>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.</li> </ol> <p>For examples of the packets generated by these flows, see Section 35.7.</p>								

#### NOTES:

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

### 35.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

**Table 35-25. 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**

Name	MODE.Exec Packet																													
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>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 &amp; LMA)</td> </tr> </table>				7	6	5	4	3	2	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	CS.D	(CS.L & LMA)
	7	6	5	4	3	2	1	0																						
0	1	0	0	1	1	0	0	1																						
1	0	0	0	0	0	0	CS.D	(CS.L & LMA)																						
Dependencies	PacketEn	Generation Scenario	Far branch, interrupt, exception, VM exit, and VM entry, if the mode changes. PSB+, and any scenario that can generate a TIP.PGE, such that the mode may have changed since the last MODE.Exec.																											
Description	<p>Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L &amp; IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.</p> <table border="1"> <thead> <tr> <th>CS.D</th> <th>(CS.L &amp; IA32_EFER.LMA)</th> <th>Addressing Mode</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>N/A</td> </tr> <tr> <td>0</td> <td>1</td> <td>64-bit mode</td> </tr> <tr> <td>1</td> <td>0</td> <td>32-bit mode</td> </tr> <tr> <td>0</td> <td>0</td> <td>16-bit mode</td> </tr> </tbody> </table> <p>MODE.Exec is sent at the time of a mode change, if PacketEn=1 at the time, or when tracing resumes, if necessary. In the former case, the MODE.Exec packet is generated along with other packets that result from the far transfer operation that changes the mode. In cases where the mode changes while PacketEn=0, the processor will send out a MODE.Exec along with the TIP.PGE when tracing resumes. The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that from the last MODE.Exec packet, if there was no PSB in between.</p>			CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode	1	1	N/A	0	1	64-bit mode	1	0	32-bit mode	0	0	16-bit mode												
CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode																												
1	1	N/A																												
0	1	64-bit mode																												
1	0	32-bit mode																												
0	0	16-bit mode																												
Application	MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.																													



## MODE.TSX Packet

Table 35-27. MODE.TSX Packet Definition

Name	MODE.TSX Packet																																		
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>TXAbort</td> <td>InTX</td> </tr> </table>									7	6	5	4	3	2	1	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0	0	TXAbort	InTX
	7	6	5	4	3	2	1	0																											
0	1	0	0	1	1	0	0	1																											
1	0	0	1	0	0	0	TXAbort	InTX																											
Dependencies	TriggerEn and ContextEn	Generation Scenario	XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if InTX changes, Asynchronous TSX Abort, PSB+																																
Description	<p>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.</p> <table border="1"> <thead> <tr> <th>TXAbort</th> <th>InTX</th> <th>Implication</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>N/A</td> </tr> <tr> <td>0</td> <td>1</td> <td>Transaction begins, or executing transactionally</td> </tr> <tr> <td>1</td> <td>0</td> <td>Transaction aborted</td> </tr> <tr> <td>0</td> <td>0</td> <td>Transaction committed, or not executing transactionally</td> </tr> </tbody> </table>								TXAbort	InTX	Implication	1	1	N/A	0	1	Transaction begins, or executing transactionally	1	0	Transaction aborted	0	0	Transaction committed, or not executing transactionally												
TXAbort	InTX	Implication																																	
1	1	N/A																																	
0	1	Transaction begins, or executing transactionally																																	
1	0	Transaction aborted																																	
0	0	Transaction committed, or not executing transactionally																																	
Application	<p>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.</p> <p>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.</p>																																		

### 35.4.2.9 TraceStop Packet

**Table 35-28. TraceStop Packet Definition**

Name	TraceStop Packet																													
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	1	0	0	0	0	0	1	1																						
Dependencies	TriggerEn && ContextEn	Generation Scenario	Taken branch with target in TraceStop IP region, MOV CR3 in TraceStop IP region, or WRMSR that sets TraceEn in TraceStop IP region.																											
Description	<p>Indicates when software has entered a user-configured TraceStop region. When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated.</p> <p>The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 35.2.4.3 for more details.</p>																													
Application	<p>If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any).</p> <p>If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.</p>																													

### 35.4.2.10 Core:Bus Ratio (CBR) Packet

**Table 35-29. CBR Packet Definition**

Name	Core:Bus Ratio (CBR) Packet																																															
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td colspan="8">Core:Bus Ratio</td> </tr> <tr> <td>3</td> <td colspan="8">Reserved</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	2	Core:Bus Ratio								3	Reserved							
	7	6	5	4	3	2	1	0																																								
0	0	0	0	0	0	0	1	0																																								
1	0	0	0	0	0	0	1	1																																								
2	Core:Bus Ratio																																															
3	Reserved																																															
Dependencies	TriggerEn	Generation Scenario	After any frequency change, on C-state wake up, PSB+, and after enabling trace packet generation.																																													
Description	Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.																																															
Application	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.																																															

## 35.4.2.11 Timestamp Counter (TSC) Packet

Table 35-30. TSC Packet Definition

Name	Timestamp Counter (TSC) Packet																																																																																			
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>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> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	1	1	0	0	1	1	SW TSC[7:0]								2	SW TSC[15:8]								3	SW TSC[23:16]								4	SW TSC[31:24]								5	SW TSC[39:32]								6	SW TSC[47:40]								7	SW TSC[55:48]							
	7	6	5	4	3	2	1	0																																																																												
0	0	0	0	1	1	0	0	1																																																																												
1	SW TSC[7:0]																																																																																			
2	SW TSC[15:8]																																																																																			
3	SW TSC[23:16]																																																																																			
4	SW TSC[31:24]																																																																																			
5	SW TSC[39:32]																																																																																			
6	SW TSC[47:40]																																																																																			
7	SW TSC[55:48]																																																																																			
Dependencies	IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop, This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.																																																																																	
Description	When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.																																																																																			
Application	TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.																																																																																			

### 35.4.2.12 Mini Time Counter (MTC) Packet

**Table 35-31. MTC Packet Definition**

Name	Mini time Counter (MTC) Packet																													
Packet Format	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 10%;"></td> <td style="width: 10%;">7</td> <td style="width: 10%;">6</td> <td style="width: 10%;">5</td> <td style="width: 10%;">4</td> <td style="width: 10%;">3</td> <td style="width: 10%;">2</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">CTC[N+7:N]</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	1	0	1	1	0	0	1	1	CTC[N+7:N]							
	7	6	5	4	3	2	1	0																						
0	0	1	0	1	1	0	0	1																						
1	CTC[N+7:N]																													
Dependencies	IA32_RTIT_CTL.MTCEn && TriggerEn	Generation Scenario	Periodic, based on the core crystal clock, or Always Running Timer (ART).																											
Description	<p>When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to (ART &gt;&gt; N) &amp; FFH. The frequency of the ART is related to the Maximum Non-Turbo frequency, and the ratio can be determined from CPUID leaf 15H, as described in Section 35.8.3. Software can select the threshold N, which determines the MTC frequency by setting the IA32_RTIT_CTL.MTCFreq field (see Section 35.2.7.2) to a supported value using the lookup enumerated by CPUID (see Section 35.3.1). See Section 35.8.3 for details on how to use the MTC payload to track TSC time.</p> <p>MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.</p> <p>There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that &gt;256 consecutive MTC packets should ever be dropped.</p>																													
Application	MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.																													

## 35.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 35-32. TMA Packet Definition

Name	TSC/MTC Alignment (TMA) Packet																																																																										
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>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> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	1	1	2	CTC[7:0]								3	CTC[15:8]								4	Reserved							0	5	FastCounter[7:0]								6	Reserved							FC[8]
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	0	1	1	1	0	0	1	1																																																																			
2	CTC[7:0]																																																																										
3	CTC[15:8]																																																																										
4	Reserved							0																																																																			
5	FastCounter[7:0]																																																																										
6	Reserved							FC[8]																																																																			
Dependencies	IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	Sent with any TSC packet.																																																																								
Description	The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 35.8.3.2 for details on how to make this calculation.																																																																										
Application	TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.																																																																										

### 35.4.2.14 Cycle Count (CYC) Packet

**Table 35-33. Cycle Count Packet Definition**

Name	Cycle Count (CYC) Packet																																															
Packet Format	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 10%;"></th> <th style="width: 10%;">7</th> <th style="width: 10%;">6</th> <th style="width: 10%;">5</th> <th style="width: 10%;">4</th> <th style="width: 10%;">3</th> <th style="width: 10%;">2</th> <th style="width: 10%;">1</th> <th style="width: 10%;">0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td colspan="5">Cycle Counter[4:0]</td> <td>Exp</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="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> </tbody> </table>				7	6	5	4	3	2	1	0	0	Cycle Counter[4:0]					Exp	1	1	1	Cycle Counter[11:5]							Exp	2	Cycle Counter[18:12]							Exp	...	... (if Exp = 1 in the previous byte)							
	7	6	5	4	3	2	1	0																																								
0	Cycle Counter[4:0]					Exp	1	1																																								
1	Cycle Counter[11:5]							Exp																																								
2	Cycle Counter[18:12]							Exp																																								
...	... (if Exp = 1 in the previous byte)																																															
Dependencies	IA32_RTIT_CTL.CYCEn && TriggerEn	Generation Scenario	Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 35.3.6 for CYC-eligible packets.																																													
Description	<p>The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length.</p> <p>If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on.</p> <p>CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 35.2.7.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 35.3.6.</p> <p>When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow.</p> <p>In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor's internal cycle counter doesn't cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen.</p> <p>When CycThresh&gt;0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone.</p> <p>When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time.</p>																																															
Application	<p>CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet.</p> <p>If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet.</p>																																															

## 35.4.2.15 VMCS Packet

Table 35-34. VMCS Packet Definition

Name	VMCS Packet																																																																															
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>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> </tbody> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	1	0	0	0	2	VMCS pointer [19:12]								3	VMCS pointer [27:20]								4	VMCS pointer [35:28]								5	VMCS pointer [43:36]								6	VMCS pointer [51:44]							
	7	6	5	4	3	2	1	0																																																																								
0	0	0	0	0	0	0	1	0																																																																								
1	1	1	0	0	1	0	0	0																																																																								
2	VMCS pointer [19:12]																																																																															
3	VMCS pointer [27:20]																																																																															
4	VMCS pointer [35:28]																																																																															
5	VMCS pointer [43:36]																																																																															
6	VMCS pointer [51:44]																																																																															
Dependencies	TriggerEn && ContextEn; Also in VMX operation.	Generation Scenario	Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 35.5).																																																																													
Description	<p>The VMCS packet provides a VMCS pointer for a decoder to determine the transition of code contexts:</p> <ul style="list-style-type: none"> <li>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).</li> <li>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.</li> <li>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.</li> </ul> <p>A VMCS packet generated before a VMCS pointer has been loaded, or after the VMCS pointer has been cleared will set all 64 bits in the VMCS pointer field.</p> <p>VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.</p>																																																																															
Application	<p>The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique.</p> <p>When a VMCS packet is encountered, a decoder should do the following:</p> <ul style="list-style-type: none"> <li>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.</li> <li>Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.</li> </ul> <p>For examples of the packets generated by these flows, see Section 35.7.</p>																																																																															

### 35.4.2.16 Overflow (OVF) Packet

Table 35-35. OVF Packet Definition

Name	Overflow (OVF) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	1	0
	1	1	1	1	1	0	0	1	1
Dependencies	TriggerEn	Generation Scenario		On resolution of internal buffer overflow					
Description	OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. If BranchEN= 1, OVF is followed by a FUP or TIP.PGE which will provide the IP at which packet generation resumes. See Section 35.3.8.								
Application	When an OVF packet is encountered, the decoder should skip to the IP given in the subsequent FUP or TIP.PGE. The cycle counter for the CYC packet will be reset at the time the OVF packet is sent. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that preceded the overflow.								

### 35.4.2.17 Packet Stream Boundary (PSB) Packet

Table 35-36. 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	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	1	1
	7	6	5	4	3	2	1	0																											
0	0	0	0	0	0	0	1	0																											
1	0	0	1	0	0	0	1	1																											
Dependencies	TriggerEn	Generation Scenario	Always follows PSB packet, separated by PSB+ packets																																
Description	PSBEND is simply a terminator for the series of “status only” (PSB+) packets that follow PSB (Section 35.3.7).																																		
Application	When a PSBEND packet is seen, the decoder should cease to treat packets as “status only”.																																		

### 35.4.2.19 Maintenance (MNT) Packet

**Table 35-38. MNT Packet Definition**

Name	Maintenance (MNT) Packet																																																																																																														
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>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>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	2	1	0	0	0	1	0	0	0	3	Payload[7:0]								4	Payload[15:8]								5	Payload[23:16]								6	Payload[31:24]								7	Payload[39:32]								8	Payload[47:40]								9	Payload[55:48]								10	Payload[63:56]							
	7	6	5	4	3	2	1	0																																																																																																							
0	0	0	0	0	0	0	1	0																																																																																																							
1	1	1	0	0	0	0	1	1																																																																																																							
2	1	0	0	0	1	0	0	0																																																																																																							
3	Payload[7:0]																																																																																																														
4	Payload[15:8]																																																																																																														
5	Payload[23:16]																																																																																																														
6	Payload[31:24]																																																																																																														
7	Payload[39:32]																																																																																																														
8	Payload[47:40]																																																																																																														
9	Payload[55:48]																																																																																																														
10	Payload[63:56]																																																																																																														
Dependencies	TriggerEn	Generation Scenario	Implementation specific.																																																																																																												
Description	This packet is generated by hardware, the payload meaning is model-specific.																																																																																																														
Application	Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.																																																																																																														

### 35.4.2.20 PAD Packet

**Table 35-39. PAD Packet Definition**

Name	PAD Packet																				
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0
	7	6	5	4	3	2	1	0													
0	0	0	0	0	0	0	0	0													
Dependencies	TriggerEn	Generation Scenario	Implementation specific																		
Description	PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.																				
Application	Ignore PAD packets.																				

## 35.4.2.21 PTWRITE (PTW) Packet

Table 35-40. PTW Packet Definition

Name	PTW Packet																																																																																																										
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>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>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	PayloadBytes		1	0	0	1	0	2	Payload[7:0]								3	Payload[15:8]								4	Payload[23:16]								5	Payload[31:24]								6	Payload[39:32]								7	Payload[47:40]								8	Payload[55:48]								9	Payload[63:56]							
	7	6	5	4	3	2	1	0																																																																																																			
0	0	0	0	0	0	0	1	0																																																																																																			
1	IP	PayloadBytes		1	0	0	1	0																																																																																																			
2	Payload[7:0]																																																																																																										
3	Payload[15:8]																																																																																																										
4	Payload[23:16]																																																																																																										
5	Payload[31:24]																																																																																																										
6	Payload[39:32]																																																																																																										
7	Payload[47:40]																																																																																																										
8	Payload[55:48]																																																																																																										
9	Payload[63:56]																																																																																																										
	<p>The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:</p> <table border="1"> <thead> <tr> <th>PayloadBytes</th> <th>Bytes of Payload</th> </tr> </thead> <tbody> <tr> <td>'00</td> <td>4</td> </tr> <tr> <td>'01</td> <td>8</td> </tr> <tr> <td>'10</td> <td>Reserved</td> </tr> <tr> <td>'11</td> <td>Reserved</td> </tr> </tbody> </table> <p>IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</p>								PayloadBytes	Bytes of Payload	'00	4	'01	8	'10	Reserved	'11	Reserved																																																																																									
PayloadBytes	Bytes of Payload																																																																																																										
'00	4																																																																																																										
'01	8																																																																																																										
'10	Reserved																																																																																																										
'11	Reserved																																																																																																										
Dependencies	TriggerEn & ContextEn & FilterEn & PTWEn	Generation Scenario	PTWRITE Instruction																																																																																																								
Description	<p>Contains the value held in the PTWRITE operand.  This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</p>																																																																																																										
Application	<p>Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.</p>																																																																																																										

### 35.4.2.22 Execution Stop (EXSTOP) Packet

**Table 35-41. EXSTOP Packet Definition**

Name	EXSTOP Packet																													
Packet Format	<table border="1" style="margin-left: 20px;"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>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> <p>IP bit indicates if a FUP will follow.</p>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	IP	1	1	0	0	0	1	0																						
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	C-state entry, P-state change, or other processor clock power-down. Includes : <ul style="list-style-type: none"> <li>▪ Entry to C-state deeper than C0.0</li> <li>▪ TM1/2</li> <li>▪ STPCLK#</li> <li>▪ Frequency change due to IA32_CLOCK_MODULATION, Turbo</li> </ul>																											
Description	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.																													
Application	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.																													

## 35.4.2.23 MWAIT Packet

Table 35-42. MWAIT Packet Definition

Name	MWAIT Packet																																																																																																											
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="8">MWAIT Hints[7:0]</td> </tr> <tr> <td>3</td> <td colspan="8">Reserved</td> </tr> <tr> <td>4</td> <td colspan="8">Reserved</td> </tr> <tr> <td>5</td> <td colspan="8">Reserved</td> </tr> <tr> <td>6</td> <td colspan="6">Reserved</td> <td colspan="2">EXT[1:0]</td> </tr> <tr> <td>7</td> <td colspan="8">Reserved</td> </tr> <tr> <td>8</td> <td colspan="8">Reserved</td> </tr> <tr> <td>9</td> <td colspan="8">Reserved</td> </tr> </tbody> </table>										7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	2	MWAIT Hints[7:0]								3	Reserved								4	Reserved								5	Reserved								6	Reserved						EXT[1:0]		7	Reserved								8	Reserved								9	Reserved							
	7	6	5	4	3	2	1	0																																																																																																				
0	0	0	0	0	0	0	1	0																																																																																																				
1	1	1	0	0	0	0	1	0																																																																																																				
2	MWAIT Hints[7:0]																																																																																																											
3	Reserved																																																																																																											
4	Reserved																																																																																																											
5	Reserved																																																																																																											
6	Reserved						EXT[1:0]																																																																																																					
7	Reserved																																																																																																											
8	Reserved																																																																																																											
9	Reserved																																																																																																											
Dependencies	TriggerEn & PwrEvtEn & ContextEn	Generation Scenario	MWAIT instruction, or I/O redirection to MWAIT, that complete without fault or VMexit.																																																																																																									
Description	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.																																																																																																											
Application	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.																																																																																																											

### 35.4.2.24 Power Entry (PWRE) Packet

**Table 35-43. PWRE Packet Definition**

Name	PWRE Packet																																															
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>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>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	1	0	2	HW	Reserved							3	Resolved Thread C-State				Resolved Thread Sub C-State			
	7	6	5	4	3	2	1	0																																								
0	0	0	0	0	0	0	1	0																																								
1	0	0	1	0	0	0	1	0																																								
2	HW	Reserved																																														
3	Resolved Thread C-State				Resolved Thread Sub C-State																																											
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition to a C-state deeper than C0.0.																																													
Description	<p>Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.</p> <p>Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core.</p> <p>If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, 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 <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B</i>) is an example of such a case.</p>																																															
Application	<p>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.</p>																																															

### 35.4.2.25 Power Exit (PWRX) Packet

Table 35-44. PWRX Packet Definition

Name	PWRX Packet																																																																										
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="4">Last Core C-State</td> <td colspan="4">Deepest Core C-State</td> </tr> <tr> <td>3</td> <td colspan="4">Reserved</td> <td colspan="4">Wake Reason</td> </tr> <tr> <td>4</td> <td colspan="8">Reserved</td> </tr> <tr> <td>5</td> <td colspan="8">Reserved</td> </tr> <tr> <td>6</td> <td colspan="8">Reserved</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1	0	2	Last Core C-State				Deepest Core C-State				3	Reserved				Wake Reason				4	Reserved								5	Reserved								6	Reserved							
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	1	0	1	0	0	0	1	0																																																																			
2	Last Core C-State				Deepest Core C-State																																																																						
3	Reserved				Wake Reason																																																																						
4	Reserved																																																																										
5	Reserved																																																																										
6	Reserved																																																																										
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition from a C-state deeper than C0.0 to C0.																																																																								
Description	<p>Indicates processor return to thread C0 from a C-state deeper than C0.0. The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B</i>. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested. The Wake Reason field is one-hot, encoded as follows:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Field</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Interrupt</td> <td>Wake due to external interrupt received.</td> </tr> <tr> <td>1</td> <td>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> </tbody> </table>			Bit	Field	Meaning	0	Interrupt	Wake due to external interrupt received.	1	Reserved		2	Store to Monitored Address	Wake due to store to monitored address.	3	HW Wake	Wake due to hardware autonomous condition, such as HDC.																																																									
Bit	Field	Meaning																																																																									
0	Interrupt	Wake due to external interrupt received.																																																																									
1	Reserved																																																																										
2	Store to Monitored Address	Wake due to store to monitored address.																																																																									
3	HW Wake	Wake due to hardware autonomous condition, such as HDC.																																																																									
Application	PWRX will always apply to the same IP as the PWRE. The time of wake can be discerned from (optional) timing packets that precede PWRX.																																																																										

## 35.5 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 <sup>1</sup>	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.PGD(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.TSX(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0	D.C.		None
27d	XEND/XRELEASE that does not clear InTX	1	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1	1		MODE.TSX(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE.TSX(InTX=0, TXAbort=1), TraceStop?
28b	XABORT(Async XAbort, or other)	0	1	1		MODE.TSX(InTX=0, TXAbort=1), TIP.PGE(BLIP)
28c	XABORT(Async XAbort, or other)	1	0	1	*TraceStop if BLIP is in a TraceStop region	MODE.TSX(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE.TSX(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	PIP(0), TraceStop?
30c	INIT (BSP)	0	1	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, PIP(0), TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0	0		FUP(NLIP), TIP.PGD()
30e	INIT (BSP)	1	0	1	* PIP if OS=1 *TraceStop if RESET.LIP is in a TraceStop region	FUP(NLIP), PIP(0), TIP.PGD, TraceStop?
30f	INIT (BSP)	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB * PIP if OS=1	FUP(NLIP), PIP(0)?, MODE.Exec?, TIP(ResetLIP)
31a	INIT (AP, goes to wait-for-SIPI)	0	D.C.	D.C.		None

**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(SIPI LIP); TraceStop?
32f	SIPI	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP(SIPI LIP)
33a	MWAIT (to C0)	D.C.	D.C.	D.C.		None
33b	MWAIT (to higher-numbered C-State, packet sent on wake)	D.C.	D.C.	D.C.	*TSC if TSCEn=1 *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

In Table 35-52, 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	D.C.	D.C.	D.C.		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	D.C.	D.C.	D.C.		See VM exit examples (16[a-z] in Table 35-50) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests C0, or monitor not armed, or VMX virtual-interrupt delivery	D.C.	D.C.	D.C.		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	D.C.	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	D.C.	D.C.	1		MWAIT(Cy), PWRE(Cx), EXSTOP(IP), FUP(CLIP)
16.5a	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	D.C.	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP, TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.5b	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	D.C.	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	D.C.	D.C.	1		MWAIT(5), PWRE(C1), EXSTOP(IP), FUP(CLIP)

**Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxE After	Other Dependencies	Packets Output
16.9a	HLT, Triple-fault shutdown, #MC with CR4.MCE=0, RSM to Cx (x>0)	D.C.	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	D.C.	D.C.			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	D.C.	0	0		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.10b	VMX abort	D.C.	D.C.	1		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	D.C.	0	0		See "RSM to Shutdown" (cases 15[def] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.11b	RSM to Shutdown	D.C.	D.C.	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)	D.C.	0	0		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.12b	INIT (BSP)	D.C.	D.C.	1		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.13a	INIT (AP, goes to Wait-for-SIPI)	D.C.	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) 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	CntxE After	Other Dependencies	Packets Output
16.13b	INIT (AP, goes to Wait-for-SIPI)	D.C.	D.C.	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	D.C.	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP, TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.14b	Hardware Duty Cycling (HDC)	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	D.C.	0	0		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.15b	VM entry to HLT or Shutdown	D.C.	D.C.	1		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STP-CLK#	D.C.	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP, TSC?, TMA?, CBR
16.16b	EIST in C0, S1/TM1/TM2, or STP-CLK#	D.C.	D.C.	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	D.C.	D.C.	D.C.		None
16.18	INTR during Cx (x>0)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x1)  See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.18	SMI during Cx (x>0)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)  See "HW Interrupt" (cases 14[a-z] in Table 35-50) for BranchEn packets that follow.

**Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEEn After	Other Dependencies	Packets Output
16.19	NMI during Cx (x>0)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)  See “HW Interrupt” (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.20	Store to monitored address during Cx (x>0)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x4)
16.22	#MC, IERR, TSC deadline timer expiration, or APIC counter under-flow during Cx (x>0)	D.C.	D.C.	D.C.	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 35-52, 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	CntxEEn After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	D.C.	D.C.	D.C.		None
16.24b	PTWRITE rm32/64, no fault	D.C.	0	0		None
16.24d	PTWRITE rm32, no fault	D.C.	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	D.C.	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	D.C.	D.C.	D.C.		See “Exception/fault” (cases 13[a-z] in Table 35-50) 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 Section 35.2.8.3, SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follow:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM

- Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
- For processors on which Intel PT and LBR use are mutually exclusive (see Section 35.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

## 35.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the “in-use” ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix “IA32\_RTIT\_” in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, where “in-use” can be determined by reading the “enable bits” in the configuration MSRs.
- Relinquish ownership of the trace configuration MSRs by clearing the “enabled bits” of those configuration MSRs.

## 35.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

- Processor timestamp counter

This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] =0.

- Core crystal clock

The ratio of the core crystal clock to timestamp counter frequency is known as P, and can be calculated as CPUID.15H:EBX[31:0] / CPUID.15H:EAX[31:0]. The frequency of the core crystal clock is fixed and lower than that of the timestamp counter. The periodic MTC packet is generated based on software-selected multiples of the crystal clock frequency. The MTC packet is expected to occur more frequently than the TSC packet.

- Processor core clock

The processor core clock frequency can vary due to P-state and thermal conditions. The CYC packet provides elapsed time as measured in processor core clock cycles relative to the last CYC packet.

A decoder can use all or some combination of these packets to track time at different resolutions throughout the trace packets.

### 35.8.3.1 Time Domain Relationships

The three domains are related by the following formula:

$$\text{TimeStampValue} = (\text{CoreCrystalClockValue} * P) + \text{AdjustedProcessorCycles} + \text{Software\_Offset};$$

The CoreCrystalClockValue can provide the coarse-grained component of the TSC value. P, or the TSC/“core crystal clock” ratio, can be derived from CPUID leaf 15H, as described in Section 35.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.



The `Software_Offsets` component includes software offsets that are factored into the timestamp value, such as `IA32_TSC_ADJUST`.

### 35.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 35.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the `CoreCrystalClockValue`. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

$(CTC_B - CTC_A)$ , where  $CTC_i = MTC_i[15:8] \ll IA32\_RTIT\_CTL.MTCFreq$  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 `MTCNext` packet by

$CTC_{Delta}[15:0] = (CTC_{Next}[15:0] - TMA.CTC[15:0])$ , where  $CTC_{Next} = MTC_{Payload} \ll IA32\_RTIT\_CTL.MTCFreq$ .

The `TMA.FastCounter` field provides the fractional component of the TSC packet into the next crystal clock cycle.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the `adjusted_processor_cycles` component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the `core:bus` ratio value given in the CBR packet, by multiplying the accumulated cycle count value by  $P1/CBR_{payload}$ .

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.

## 19. 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: Minor typo corrections.

## CHAPTER 37

# ENCLAVE ACCESS CONTROL AND DATA STRUCTURES

---

### 37.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT

When an enclave is created, it has a range of linear addresses to which the processor applies enhanced access control. This range is called the ELRANGE (see Section 36.3). When an enclave generates a memory access, the existing IA32 segmentation and paging architecture are applied. Additionally, linear addresses inside the ELRANGE must map to an EPC page otherwise when an enclave attempts to access that linear address a fault is generated.

The EPC pages need not be physically contiguous. System software allocates EPC pages to various enclaves. Enclaves must abide by OS/VMM imposed segmentation and paging policies. OS/VMM-managed page tables and extended page tables provide address translation for the enclave pages. Hardware requires that these pages are properly mapped to EPC (any failure generates an exception).

Enclave entry must happen through specific enclave instructions:

- ENCLU[EENTER], ENCLU[ERESUME].

Enclave exit must happen through specific enclave instructions or events:

- ENCLU[EEXIT], Asynchronous Enclave Exit (AEX).

Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations. As an example a read of an enclave page may result in the return of all one's or return of cyphertext of the cache line. Writing to an enclave page may result in a dropped write or a machine check at a later time. The processor will provide the protections as described in Section 37.4 and Section 37.5 on such accesses.

### 37.2 TERMINOLOGY

A memory access to the ELRANGE and initiated by an instruction executed by an enclave is called a Direct Enclave Access (Direct EA).

Memory accesses initiated by certain Intel® SGX instruction leaf functions such as ECREATE, EADD, EDBGD, 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<sup>1</sup> (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
EREPOR	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	<p>The RESERVED field consists of the following:</p> <ul style="list-style-type: none"> <li>▪ EID: An 8 byte Enclave Identifier. Its location is implementation specific.</li> <li>▪ PAD: A 352 bytes padding pattern from the Signature (used for key derivation strings). It's location is implementation specific.</li> <li>▪ VIRTCHILDCNT: An 8 byte Count of virtual children that have been paged out by a VMM. Its location is implementation specific.</li> <li>▪ ENCLAVECONTEXT: An 8 byte Enclave context pointer. Its location is implementation specific.</li> <li>▪ ISVFAMILYID: A 16 byte value assigned to identify the family of products the enclave belongs to.</li> <li>▪ ISVEXTPRODID: A 16 byte value assigned to identify the product identity of the enclave.</li> <li>▪ The remaining 3226 bytes are reserved area.</li> </ul> <p>The entire 3836 byte field must be cleared prior to executing ECREATE.</p>

### 37.7.1 ATTRIBUTES

The ATTRIBUTES data structure is comprised of bit-granular fields that are used in the SECS, the REPORT and the KEYREQUEST structures. CPUID.(EAX=12H, ECX=1) enumerates a bitmap of permitted 1-setting of bits in ATTRIBUTES.

**Table 37-3. Layout of ATTRIBUTES Structure**

Field	Bit Position	Description
INIT	0	This bit indicates if the enclave has been initialized by EINIT. It must be cleared when loaded as part of ECREATE. For EREPORT instruction, TARGET_INFO.ATTRIBUTES[ENIT] must always be 1 to match the state after EINIT has initialized the enclave.
DEBUG	1	If 1, the enclave permit debugger to read and write enclave data using EDBGDR 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
EINITOKEN_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 processor is currently executing an enclave in the context of this TCS.
FLAGS	8	8	The thread's execution flags (see Section 37.8.1).
OSSA	16	8	Offset of the base of the State Save Area stack, relative to the enclave base. Must be page aligned.
CSSA	24	4	Current slot index of an SSA frame, cleared by EADD and EACCEPT.
NSSA	28	4	Number of available slots for SSA frames.
OENTRY	32	8	Offset in enclave to which control is transferred on EENTER relative to the base of the enclave.
AEP	40	8	The value of the Asynchronous Exit Pointer that was saved at EENTER time.
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 ODH information	The size of XSAVE region in SSA is derived from the enclave’s support of the collection of processor extended states that would be managed by XSAVE. The enablement of those processor extended state components in conjunction with CPUID leaf ODH information determines the XSAVE region size in SSA.
Pad	End of XSAVE region	Chosen by enclave writer	Ensure the end of GPRSGX region is aligned to the end of a 4KB page.
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: <ul style="list-style-type: none"> <li>• Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM.</li> <li>• Conditionally supported exception: <ul style="list-style-type: none"> <li>– #PF, #GP if SECS.MISCSELECT.EXITINFO = 1.</li> </ul> </li> </ul>

### 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.EXITINFO = 1.
#PF	14	Page fault exception. Only reported if SECS.MISCSELECT.EXITINFO = 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 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 <sup>1</sup>	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:

$$q1 = \text{floor}(\text{Signature}^2 / \text{Modulus});$$

$$q2 = \text{floor}((\text{Signature}^3 - q1 * \text{Signature} * \text{Modulus}) / \text{Modulus});$$

SIGSTRUCT must be page aligned

In column 5 of Table 37-19, '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 06000000E1000000000010000000000H	Y
VENDOR	16	4	Intel Enclave: 00008086H Non-Intel Enclave: 00000000H	Y
DATE	20	4	Build date is yyyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31	Y
HEADER2	24	16	Must be byte stream 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.
CPUSVNL	192	16	N	Launch Enclave's CPUSVN.
ISVPRODID	208	02	N	Launch Enclave's ISVPRODID.
ISVSVNL	210	02	N	Launch Enclave's ISVSVN.
RESERVED	212	24	N	Reserved.
MASKEDMISCSELECT	236	4		Launch Enclave's MASKEDMISCSELECT: set by the LE to the resolved MISCSELECT value, used by EGETKEY (after applying KEYREQUEST's masking).
MASKEDATTRIBUTES	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.ISVEXTNPRODID.
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	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
CONFIGSVN	260	2	Value provided by SW to indicate expected SVN of enclave's post EINIT configuration.
RESERVED	262	42	Zero.
ISVFAMILYID	304	16	The value of SECS.ISVFAMILYID.

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	2	Identifies the Key Required.
KEYPOLICY	2	2	Identifies which inputs are required to be used in the key derivation.
ISVSVN	4	2	The ISV security version number that will be used in the key derivation.
RESERVED	6	2	Must be zero.
CPUSVN	8	16	The security version number of the processor used in the key derivation.
ATTRIBUTEMASK	24	16	A mask defining which ATTRIBUTES bits will be included in key derivation.
KEYID	40	32	Value for key wear-out protection.
MISCMASK	72	4	A mask defining which MISCSELECT bits will be included in key derivation.

**Table 37-23. Layout of KEYREQUEST Data Structure**

Field	OFFSET (Bytes)	Size (Bytes)	Description
CONFIGSVN	76	2	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's ISVPRODID value.
CONFIGID	3	If 1, derive key using the enclave's CONFIGID value.
ISVFAMILYID	4	If 1, derive key using the enclave ISVFAMILYID value.
ISVEXTPRODID	5	If 1, derive key using enclave's ISVEXTPRODID value.
RESERVED	15:6	Must be zero.

## 37.18 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	8	Version Slot 0
Slot 1	8	8	Version Slot 1
...			
Slot 511	4088	8	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.

## 20. 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: Update to Table 38-1 "Illegal Instructions Inside an Enclave". Update to Section 38.6.4 "Executions of INT1 and INT3 Inside an Enclave". Minor typo corrections.

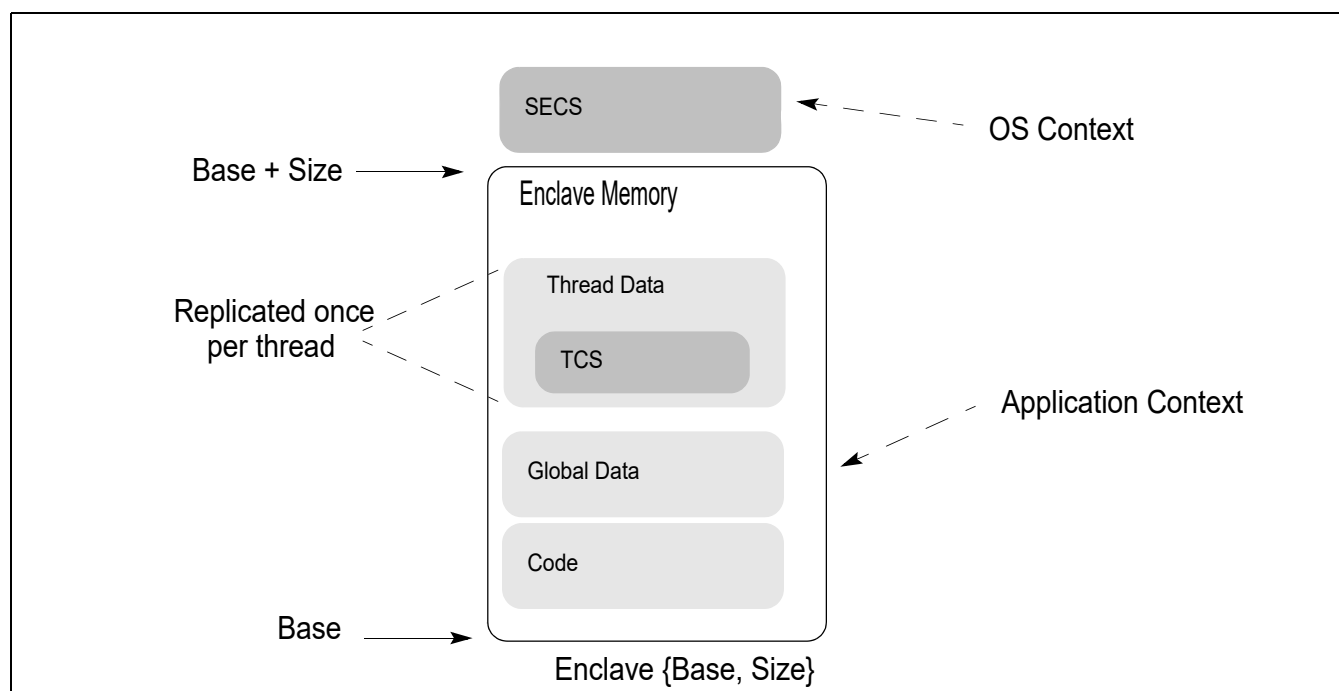
## CHAPTER 38 ENCLAVE OPERATION

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:

## ENCLAVE OPERATION

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.

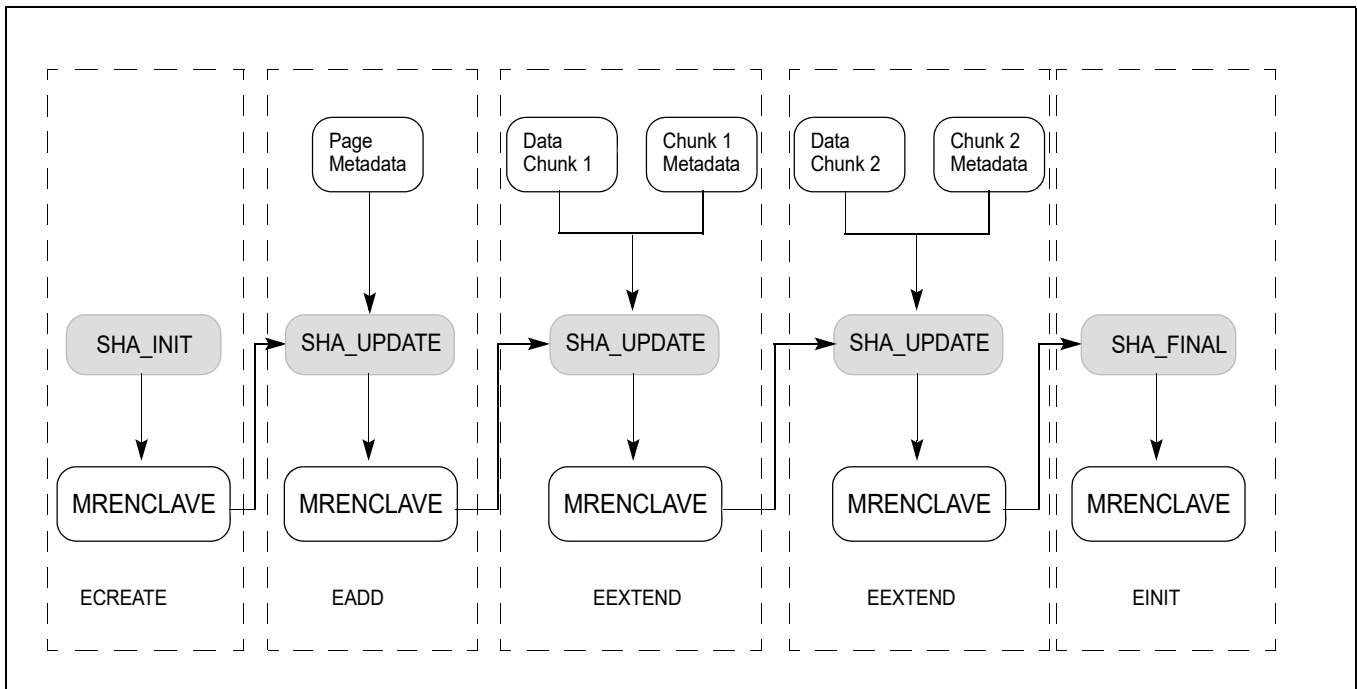


Figure 38-2. Measurement Flow of Enclave Build Process

### 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.

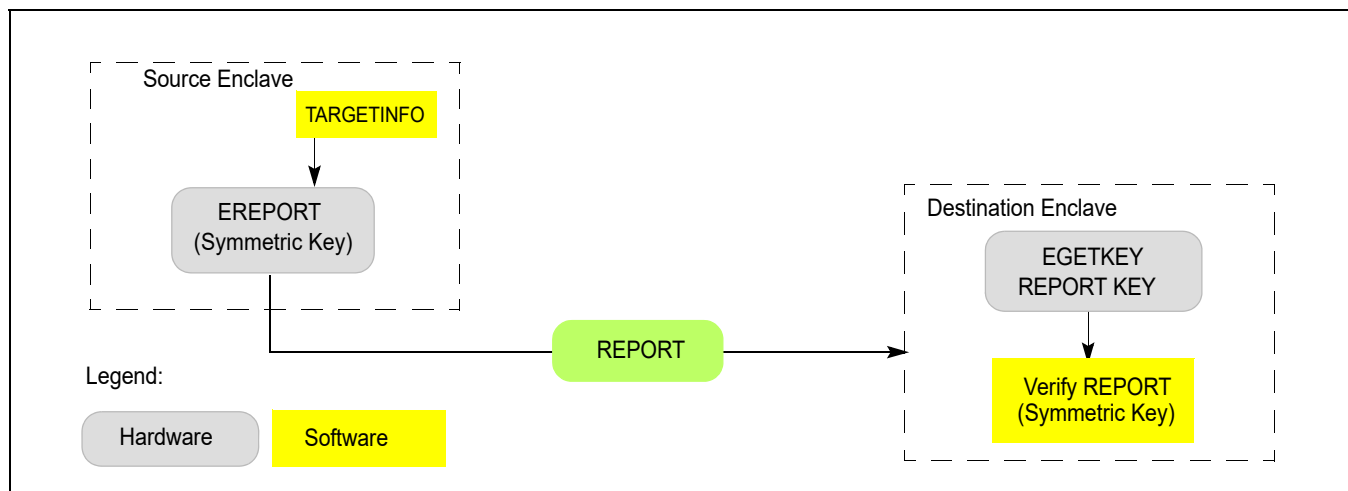


Figure 38-3. SGX Local Attestation

## 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 shutdown".
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 allocates 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 shutdown.
  - 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 by 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 (i.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 of 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 of 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 of 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 of these instructions inside an enclave results in #UD.

**Table 38-1. Illegal Instructions Inside an Enclave**

Instructions	Result	Comment
CPUID, GETSEC, RDPMMC, 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 Executions of INT1 and INT3 Inside an Enclave

The INT1 and INT3 instructions are legal inside an enclave, however, their behavior inside an enclave differs from that 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).



## 21. Updates to Chapter 39, Volume 3D

Change bars show changes to Chapter 39 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Change to chapter: Minor typo corrections.

## CHAPTER 39

# ENCLAVE EXITING EVENTS

Certain events, such as exceptions and interrupts, incident to (but asynchronous with) enclave execution may cause control to transition outside of enclave mode. (Most of these also cause a change of privilege level.) To protect the integrity and security of the enclave, the processor will exit the enclave (and enclave mode) before invoking the handler for such an event. For that reason, such events are called **enclave-exiting events (EEE)**; EEEs include external interrupts, non-maskable interrupts, system-management interrupts, exceptions, and VM exits.

The process of leaving an enclave in response to an EEE is called an **asynchronous enclave exit (AEX)**. To protect the secrecy of the enclave, an AEX saves the state of certain registers within enclave memory and then loads those registers with fixed values called **synthetic state**.

### 39.1 COMPATIBLE SWITCH TO THE EXITING STACK OF AEX

AEXs load registers with a pre-determined synthetic state. These registers may be later pushed onto the appropriate stack in a form as defined by the enclave-exiting event. To allow enclave execution to resume after the invoking handler has processed the enclave exiting event, the asynchronous enclave exit loads the address of trampoline code outside of the enclave into RIP. This trampoline code eventually returns to the enclave by means of an ENCLU(ERESUME) leaf function. Prior to exiting the enclave the RSP and RBP registers are restored to their values prior to enclave entry.

The stack to be used is chosen using the same rules as for non-SGX mode:

- If there is a privilege level change, the stack will be the one associated with the new ring.
- If there is no privilege level change, the current application stack is used.
- If the IA-32e IST mechanism is used, the exit stack is chosen using that method.

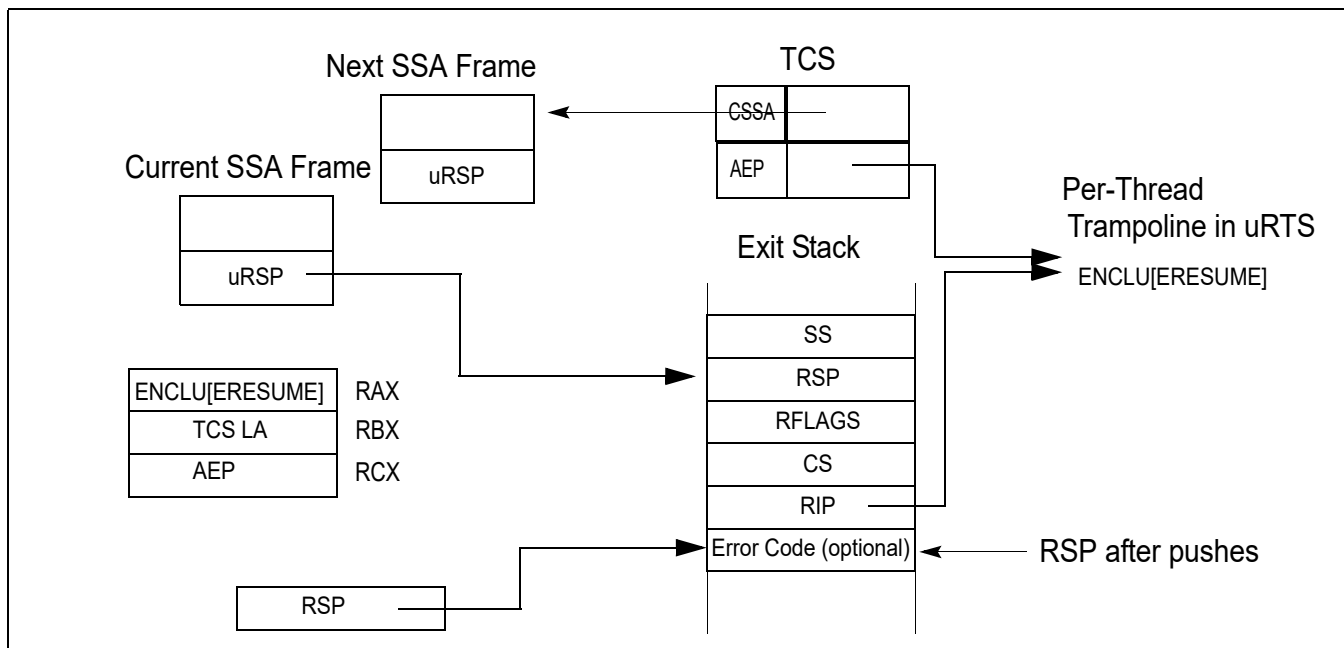


Figure 39-1. Exit Stack Just After Interrupt with Stack Switch



In all cases, the choice of exit stack and the information pushed onto it is consistent with non-SGX operation. Figure 39-1 shows the Application and Exiting Stacks after an exit with a stack switch. An exit without a stack switch uses the Application Stack. The ERESUME leaf index value is placed into RAX, the TCS pointer is placed in RBX and the AEP (see below) is placed into RCX to facilitate resuming the enclave after the exit.

Upon an AEX, the AEP (Asynchronous Exit Pointer) is loaded into the RIP. The AEP points to a trampoline code sequence which includes the ERESUME instruction that is later used to reenter the enclave.

The following bits of RFLAGS are cleared before RFLAGS is pushed onto the exit stack: CF, PF, AF, ZF, SF, OF, RF. The remaining bits are left unchanged.

## 39.2 STATE SAVING BY AEX

The State Save Area holds the processor state at the time of an AEX. To allow handling events within the enclave and re-entering it after an AEX, the SSA can be a stack of multiple SSA frames as illustrated in Figure 39-2.

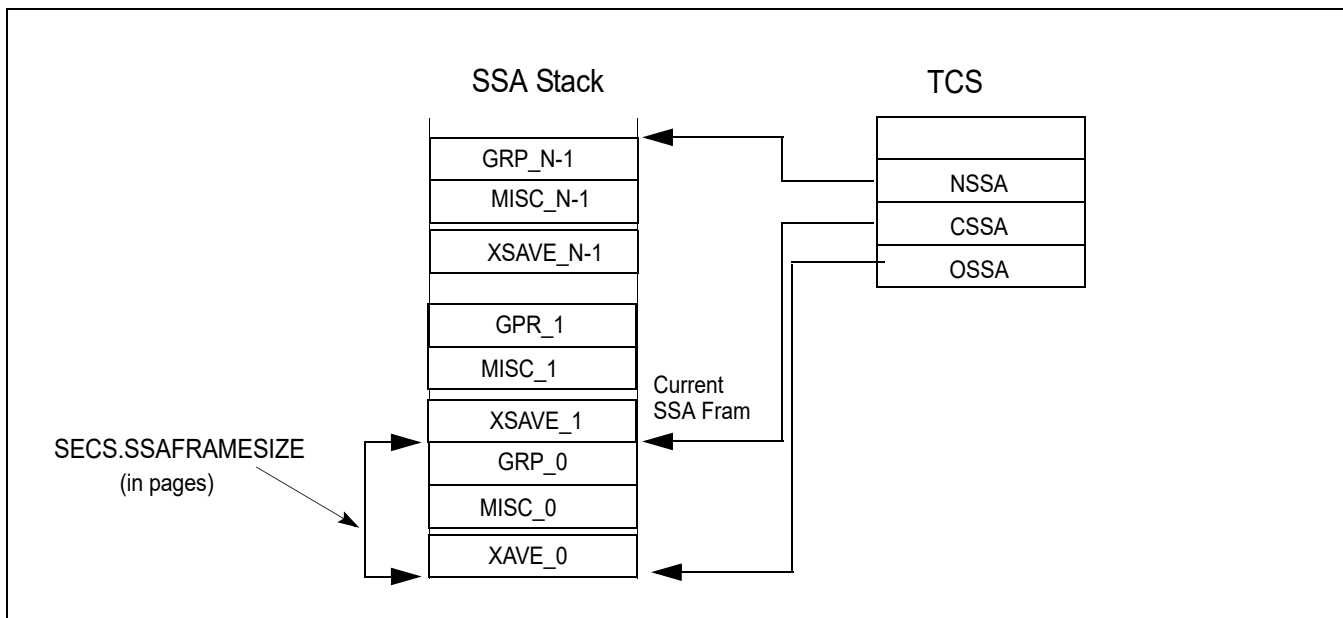


Figure 39-2. The SSA Stack

The location of the SSA frames to be used is controlled by the following variables in the TCS and the SECS:

- Size of a frame in the State Save Area (SECS.SSAFRAMESIZE): This defines the number of 4-KByte pages in a single frame in the State Save Area. The SSA frame size must be large enough to hold the GPR state, the XSAVE state, and the MISC state.
- Base address of the enclave (SECS.BASEADDR): This defines the enclave's base linear address from which the offset to the base of the SSA stack is calculated.
- Number of State Save Area Slots (TCS.NSSA): This defines the total number of slots (frames) in the State Save Area stack.
- Current State Save Area Slot (TCS.CSSA): This defines the slot to use on the next exit.
- State Save Area Offset (TCS.OSSA): This defines the offset of the base address of a set of State Save Area slots from the enclave's base address.

When an AEX occurs, hardware selects the SSA frame to use by examining TCS.CSSA. Processor state is saved into the SSA frame (see Section 39.4) and loaded with a synthetic state (as described in Section 39.3.1) to avoid leaking secrets, RSP and RBP are restored to their values prior to enclave entry, and TCS.CSSA is incremented. As will be described later, if an exception takes the last slot, it will not be possible to reenter the enclave to handle the

exception from within the enclave. A subsequent ERESUME restores the processor state from the current SSA frame and frees the SSA frame.

The format of the XSAVE section of SSA is identical to the format used by the XSAVE/XRSTOR instructions. On EENTER, CSSA must be less than NSSA, ensuring that there is at least one State Save Area slot available for exits. If there is no free SSA frame when executing EENTER, the entry will fail.

## 39.3 SYNTHETIC STATE ON ASYNCHRONOUS ENCLAVE EXIT

### 39.3.1 Processor Synthetic State on Asynchronous Enclave Exit

Table 39-1 shows the synthetic state loaded on AEX. The values shown are the lower 32 bits when the processor is in 32 bit mode and 64 bits when the processor is in 64 bit mode.

**Table 39-1. GPR, x87, SSE Synthetic States on Asynchronous Enclave Exit**

Register	Value
RAX	3 (ENCLU[3] is ERESUME).
RBX	Pointer to TCS of interrupted enclave thread.
RCX	AEP of interrupted enclave thread.
RDX, RSI, RDI	0.
RSP	Restored from SSA.uRSP.
RBP	Restored from SSA.uRBP.
R8-R15	0 in 64-bit mode; unchanged in 32-bit mode.
RIP	AEP of interrupted enclave thread.
RFLAGS	CF, PF, AF, ZF, SF, OF, RF bits are cleared. All other bits are left unchanged.
x87/SSE State	Unless otherwise listed here, all x87 and SSE state are set to the INIT state. The INIT state is the state that would be loaded by the XRSTOR instruction with bits 1:0 both set in the requested feature bitmask (RFBM), and both clear in XSTATE_BV the XSAVE header.
FCW	On #MF exception: set to 037EH. On all other exits: set to 037FH.
FSW	On #MF exception: set to 8081H. On all other exits: set to 0H.
MXCSR	On #XM exception: set to 1F01H. On all other exits: set to 1FBOH.
CR2	If the event that caused the AEX is a #PF, and the #PF does not directly cause a VM exit, then the low 12 bits are cleared. If the #PF leads directly to a VM exit, CR2 is not updated (usual IA behavior). Note: The low 12 bits are not cleared if a #PF is encountered during the delivery of the EEE that caused the AEX. This is because the #PF was not the EEE.
FS, GS	Restored to values as of most recent EENTER/ERESUME.

### 39.3.2 Synthetic State for Extended Features

When CR4.OSXSAVE = 1, extended features (those controlled by XCR0[63:2]) are set to their respective INIT states when this corresponding bit of SECS.XFRM is set. The INIT state is the state that would be loaded by the XRSTOR instruction had the instruction mask and the XSTATE\_BV field of the XSAVE header each contained the value XFRM. (When the AEX occurs in 32-bit mode, those features that do not exist in 32-bit mode are unchanged.)

### 39.3.3 Synthetic State for MISC Features

State represented by SECS.MISCSELECT might also be overridden by synthetic state after it has been saved into the SSA. State represented by MISCSELECT[0] is not overridden but if the exiting event is a page fault then lower 12 bits of CR2 are cleared.

## 39.4 AEX FLOW

On Enclave Exiting Events (interrupts, exceptions, VM exits or SMIs), the processor state is securely saved inside the enclave, a synthetic state is loaded and the enclave is exited. The EEE then proceeds in the usual exit-defined fashion. The following sections describes the details of an AEX:

1. The exact processor state saved into the current SSA frame depends on whether the enclave is a 32-bit or a 64-bit enclave. 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 stored. The upper 32 bits of the legacy registers and the 64-bit registers (R8 ... R15) are not stored.

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 stored.

The state of those extended features specified by SECS.ATTRIBUTES.XFRM are stored into the XSAVE area of the current SSA frame. The layout of the x87 and XMM portions (the 1st 512 bytes) depends on the current values of IA32\_EFER.LMA and CS.L:

If IA32\_EFER.LMA = 0 || CS.L = 0, the same format (32-bit) that XSAVE/FXSAVE uses with these values.

If IA32\_EFER.LMA = 1 && CS.L = 1, the same format (64-bit) that XSAVE/FXSAVE uses with these values when REX.W = 1.

The cause of the AEX is saved in the EXITINFO field. See Table 37-9 for details and values of the various fields.

The state of those miscellaneous features (see Section 37.7.2) specified by SECS.MISCSELECT are stored into the MISC area of the current SSA frame.

2. Synthetic state is created for a number of processor registers to present an opaque view of the enclave state. Table 39-1 shows the values for GPRs, x87, SSE, FS, GS, Debug and performance monitoring on AEX. The synthetic state for other extended features (those controlled by XCR0[62:2]) is set to their respective INIT states when their corresponding bit of SECS.ATTRIBUTES.XFRM is set. The INIT state is that state as defined by the behavior of the XRSTOR instruction when HEADER.XSTATE\_BV[n] is 0. Synthetic state of those miscellaneous features specified by SECS.MISCSELECT depends on the miscellaneous feature. There is no synthetic state required for the miscellaneous state controlled by SECS.MISCSELECT[0].
3. Any code and data breakpoints that were suppressed at the time of enclave entry are unsuppressed when exiting the enclave.
4. RFLAGS.TF is set to the value that it had at the time of the most recent enclave entry (except for the situation that the entry was opt-in for debug; see Section 42.2). In the SSA, RFLAGS.TF is set to 0.
5. RFLAGS.RF is set to 0 in the synthetic state. In the SSA, the value saved is the same as what would have been saved on stack in the non-SGX case (architectural value of RF). Thus, AEXs due to interrupts, traps, and code breakpoints save RF unmodified into SSA, while AEXs due to other faults save RF as 1 in the SSA.  
If the event causing AEX happened on intermediate iteration of a REP-prefixed instruction, then RF=1 is saved on SSA, irrespective of its priority.
6. Any performance monitoring activity (including PEBS) or profiling activity (LBR, Tracing using Intel PT) on the exiting thread that was suppressed due to the enclave entry on that thread is unsuppressed. Any counting that had been demoted from AnyThread counting to MyThread counting (on one logical processor) is promoted back to AnyThread counting.

## 39.4.1 AEX Operational Detail

### Temp Variables in AEX Operational Flow

Name	Type	Size (bits)	Description
TMP_RIP	Effective Address	32/64	Address of instruction at which to resume execution on ERESUME.
TMP_MODE64	binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_BRANCH_RECORD	LBR Record	2x64	From/To address to be pushed onto LBR stack.

The pseudo code in this section describes the internal operations that are executed when an AEX occurs in enclave mode. These operations occur just before the normal interrupt or exception processing occurs.

(\* Save RIP for later use \*)

TMP\_RIP = Linear Address of Resume RIP

(\* Is the processor in 64-bit mode? \*)

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

(\* Save all registers, When saving EFLAGS, the TF bit is set to 0 and the RF bit is set to what would have been saved on stack in the non-SGX case \*)

IF (TMP\_MODE64 = 0)

THEN

Save EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EFLAGS, EIP into the current SSA frame using CR\_GPR\_PA; (\* see Table 40-5 for list of CREGs used to describe internal operation within Intel SGX \*)

SSA.RFLAGS.TF ← 0;

ELSE (\* TMP\_MODE64 = 1 \*)

Save RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15, RFLAGS, RIP into the current SSA frame using CR\_GPR\_PA;

SSA.RFLAGS.TF ← 0;

FI;

Save FS and GS BASE into SSA using CR\_GPR\_PA;

(\* store XSAVE state into the current SSA frame's XSAVE area using the physical addresses that were determined and cached at enclave entry time with CR\_XSAVE\_PAGE\_i. \*)

For each XSAVE state i defined by (SECS.ATTRIBUTES.XFRM[i] = 1, destination address cached in CR\_XSAVE\_PAGE\_i)

SSA.XSAVE.i ← XSAVE\_STATE\_i;

(\* Clear bytes 8 to 23 of XSAVE\_HEADER, i.e. the next 16 bytes after XHEADER\_BV \*)

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[191:64] ← 0;

(\* Clear bits in XHEADER\_BV[63:0] that are not enabled in ATTRIBUTES.XFRM \*)

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[63:0] ←

CR\_XSAVE\_PAGE\_0.XHEADER\_BV[63:0] & SECS(CR\_ACTIVE\_SECS).ATTRIBUTES.XFRM;

Apply synthetic state to GPRs, RFLAGS, extended features, etc.

(\* Restore the RSP and RBP from the current SSA frame's GPR area using the physical address that was determined and cached at enclave entry time with CR\_GPR\_PA. \*)

RSP ← CR\_GPR\_PA.URSP;

RBP ← CR\_GPR\_PA.URBP;

## ENCLAVE EXITING EVENTS

(\* Restore the FS and GS \*)

```
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;
```

(\* Examine exception code and update enclave internal states\*)

```
exception_code ← Exception or interrupt vector;
```

(\* Indicate the exit reason in SSA \*)

```
IF (exception_code = (#DE OR #DB OR #BP OR #BR OR #UD OR #MF OR #AC OR #XM ))
```

```
THEN
```

```
    CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
```

```
    IF (exception_code = #BP)
```

```
        THEN CR_GPR_PA.EXITINFO.EXIT_TYPE ← 6;
```

```
        ELSE CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
```

```
    FI;
```

```
    CR_GPR_PA.EXITINFO.VALID ← 1;
```

```
ELSE IF (exception_code is #PF or #GP )
```

```
THEN
```

```
(* Check SECS.MISCSELECT using CR_ACTIVE_SECS *)
```

```
IF (SECS.MISCSELECT[0] is set)
```

```
    THEN
```

```
        CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
```

```
        CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
```

```
        IF (exception_code is #PF)
```

```
            THEN
```

```
                SSA.MISC.EXINFO. MADDR ← CR2;
```

```
                SSA.MISC.EXINFO.ERRCD ← PFEC;
```

```
                SSA.MISC.EXINFO.RESERVED ← 0;
```

```
        ELSE
```

```
            SSA.MISC.EXINFO. MADDR ← 0;
```

```
            SSA.MISC.EXINFO.ERRCD ← GPEC;
```

```
            SSA.MISC.EXINFO.RESERVED ← 0;
```

```
        FI;
```

```
        CR_GPR_PA.EXITINFO.VALID ← 1;
```

```
    FI;
```

```
ELSE
```

```
    CR_GPR_PA.EXITINFO.VECTOR ← 0;
```

```
    CR_GPR_PA.EXITINFO.EXIT_TYPE ← 0
```

```
    CR_GPR_PA.REASON.VALID ← 0;
```

```
FI;
```

(\* Execution will resume at the AEP \*)

```
RIP ← CR_TCS_PA.AEP;
```

(\* Set EAX to the ERESUME leaf index \*)

```
EAX ← 3;
```

```
(* Put the TCS LA into RBX for later use by ERESUME *)
RBX ← CR_TCS_LA;
```

```
(* Put the AEP into RCX for later use by ERESUME *)
RCX ← CR_TCS_PA.AEP;
```

```
(* Increment the SSA frame # *)
CR_TCS_PA.CSSA ← CR_TCS_PA.CSSA + 1;
```

```
(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
  THEN XCR0 ← CR_SAVE_XCR0; FI;
```

Un-suppress all code breakpoints that are outside ELRANGE

```
(* Update the thread context to show not in enclave mode *)
CR_ENCLAVE_MODE ← 0;
```

```
(* Assure consistent translations. *)
Flush linear context including TLBs and paging-structure caches
```

```
IF (CR_DBGOPTIN = 0)
  THEN
    Un-suppress all breakpoints that overlap ELRANGE
    (* Clear suppressed breakpoint matches *)
    Restore suppressed breakpoint matches
    (* Restore TF *)
    RFLAGS.TF ← CR_SAVE_TF;
    Un-suppress monitor trap flag;
    Un-suppress branch recording facilities;
    Un-suppress all suppressed performance monitoring activity;
    Promote any sibling-thread counters that were demoted from AnyThread to MyThread during enclave
    entry back to AnyThread;
  FI;
```

```
IF the "monitor trap flag" VM-execution control is 1
  THEN Pend MTF VM Exit at the end of exit; FI;
```

```
(* Clear low 12 bits of CR2 on #PF *)
IF (Exception code is #PF)
  THEN CR2 ← CR2 & ~0xFFF; FI;
```

```
(* end_of_flow *)
(* Execution continues with normal event processing. *)
```



## 22. Updates to Chapter 40, Volume 3D

Change bars show changes to Chapter 40 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Typo corrections.



## 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)	
EDBGGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
ERDINFO	010H (In)	RDINFO (In, EA*)	EPCPAGE (In, EA)	
ETRACKC	011H (In)		EPCPAGE (In, EA)	
ELDBC	012H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDUC	013H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)

EA: Effective Address

### 40.1.2 ENCLU Register Usage Summary

Table 40-2 summarizes the implicit register usage of user mode enclave instructions.

**Table 40-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions**

Instr. Leaf	EAX	RBX	RCX	RDY
EReport	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGetKey	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EEnter	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
EResume	03H (In)	TCS (In, EA)	AEP (In, EA)	
EExit	04H (In)	Target (In, EA)	Current AEP (Out)	
EAccept	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EModPE	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EAcceptCopy	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EA: Effective Address				

### 40.1.3 ENCLV Register Usage Summary

Table 40-3 summarizes the implicit register usage of virtualization operation enclave instructions.

**Table 40-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions**

Instr. Leaf	EAX	RBX	RCX	RDY
EDECvirtChild	00H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
EINCvirtChild	01H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
ESETCONTEXT	02H (In)		EPCPAGE (In, EA)	Context Value (In, EA)
EA: Effective Address				

### 40.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 40-4 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

**Table 40-4. Error or Information Codes for Intel® SGX Instructions**

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_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, EINCVRTCHILD
SGX_PG_NONEPC	26	ERDINFO
SGX_TRACK_NOT_REQUIRED	27	ETRACKC
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

### 40.1.5 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 40-5 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

**Table 40-5. List of Internal CREG**

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_EL RANGE	128	LP
CR_SAVE_TF	1	LP
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP

Table 40-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNEREPOCH	128	PACKAGE
CR_SAVE_XCRO	64	LP
CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE

## 40.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leaves to concurrently operate on the same EPC page.
  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX\_EPC\_PAGE\_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

### 40.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.
- **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.
- **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EEXTEND 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
EDBGD	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	
EDBGDR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGWR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDECVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EENTERTCS	SECS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EEXIT			Concurrent		Concurrent		Concurrent	
EEXTEND	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINCVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINIT	SECS	[DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	
ELDB/ELDU	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EDLBC/ELDUC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EMODPE	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EMODPR	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CON FLICT	Concurrent		Concurrent	
EPA	VA	[DS:RCX]	Concurrent		Concurrent		Concurrent	

**Table 40-7. Additional Concurrency Restrictions**

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREMOVE	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
ERESUME	TCS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
ESETCONTEXT	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ETRACK	SECS	[DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT <sup>1</sup>
ETRACKC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	Implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT <sup>1</sup>
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	NA	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
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 an invalid leaf number)

THEN #GP(0); FI;

IF CR0.PG = 0  
THEN #GP(0); FI;

(\* DS must not be an expanded down segment \*)  
IF not in 64-bit mode and DS.Type is expand-down data  
THEN #GP(0); FI;

Jump to leaf specific flow

**Flags Affected**

See individual leaf functions

**Protected Mode Exceptions**

#UD                    If any of the LOCK/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	NA	This instruction is used to execute non-privileged Intel SGX leaf functions.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
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 OF 01 C0 ENCLV	NP	V/V	NA	This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
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 IA32\_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 an invalid leaf number)  
THEN #GP(0); FI;

IF CR0.PG = 0  
THEN #GP(0); FI;

(\* DS must not be an expanded down segment \*)  
IF not in 64-bit mode and DS.Type is expand-down data  
THEN #GP(0); FI;

Jump to leaf specific flow

### Flags Affected

See individual leaf functions.

### Protected Mode Exceptions

#UD	If any of the LOCK/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 misconfigured SECINFO flags\*)  
IF (SCRATCH\_SECINFO reserved fields are not zero or

```
!(SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS)
THEN #GP(0); FI;
```

(\* Check the EPC page for concurrency \*)

```
IF (EPC page is not available for EADD)
THEN
  IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
  THEN
    VMCS.Exit_reason ← SGX_CONFLICT;
    VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
    VMCS.Exit_qualification.error ← 0;
    VMCS.Guest-physical_address ← << translation of DS:RCX produced by paging >>;
    VMCS.Guest-linear_address ← DS:RCX;
    Deliver VMEXIT;
  ELSE
    #GP(0);
  FI;
FI;
```

```
IF (EPCM(DS:RCX).VALID ≠ 0)
THEN #PF(DS:RCX); FI;
```

(\* Check the SECS for concurrency \*)

```
IF (SECS is not available for EADD)
THEN #GP(0); FI;
```

```
IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
THEN #PF(DS:TMP_SECS); FI;
```

(\* Copy 4KBytes from source page to EPC page\*)

```
DS:RCX[32767:0] ← DS:TMP_SRCPGE[32767:0];
```

```
CASE (SCRATCH_SECINFO.FLAGS.PT)
```

```
PT_TCS:
```

```
IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
```

```
IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
```

```
((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
```

```
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 SECFINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPT-COPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

### EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

### EAUG Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

### Concurrency Restrictions

**Table 40-10. Base Concurrency Restrictions of EAUG**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EAUG	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Table 40-11. Additional Concurrency Restrictions of EAUG

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EAUG	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Concurrent		Concurrent	

## Operation

### Temp Variables in EAUG Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

IF (DS:RBX is not 32Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

TMP\_SECS ← DS:RBX.SECS;  
TMP\_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 \*)

## SGX INSTRUCTION REFERENCES

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;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] ← DS:TMP_SRCPAGE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
    THEN #GP(0); FI;

(* 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 0FFFFFFF00000000H) )
    THEN #GP(0); FI;

IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
    THEN #GP(0); FI;

IF ( ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )

```

```
THEN #GP(0); FI;
```

```
(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
```

```
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
```

```
THEN #GP(0); FI;
```

```
(* Ensure base address of an enclave is aligned on size*)
```

```
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) ) )
```

```
THEN #GP(0); FI;
```

```
(* Ensure the SECS does not have any unsupported attributes*)
```

```
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
```

```
THEN #GP(0); FI;
```

```
IF ( DS:TMP_SECS reserved fields are not zero)
```

```
THEN #GP(0); FI;
```

```
(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
```

```
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠ 0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0) )
```

```
THEN #GP(0); FI;
```

```
Clear DS:TMP_SECS to Uninitialized;
```

```
DS:TMP_SECS.MRENCLAVE ← SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
```

```
DS:TMP_SECS.ISVSVN ← 0;
```

```
DS:TMP_SECS.ISVPRODID ← 0;
```

```
(* Initialize hash updates etc*)
```

```
Initialize enclave's MRENCLAVE update counter;
```

```
(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
```

```
TMPUPDATEFIELD[63:0] ← 0045544145524345H; // "ECREATE"
```

```
TMPUPDATEFIELD[95:64] ← DS:TMP_SECS.SSAFRAMESIZE;
```

```
TMPUPDATEFIELD[159:96] ← DS:TMP_SECS.SIZE;
```

```
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 *)
```

## SGX INSTRUCTION REFERENCES

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 EDBGD**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGD	Target [DS:RCX]	Shared	#GP	

**Table 40-19. Additional Concurrency Restrictions of EDBGD**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

**Temp Variables in EDBGD Operational Flow**

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP\_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )  
 THEN #GP(0); FI;

IF ( (TMP\_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing 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;
    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)	<p>If the address in RCS violates DS limit or access rights.</p> <p>If DS segment is unusable.</p> <p>If RCX points to a memory location not 4Byte-aligned.</p> <p>If the address in RCX points to a page belonging to a non-debug enclave.</p> <p>If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.</p> <p>If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.</p>
--------	--

## SGX INSTRUCTION REFERENCES

#PF(error code)    If a page fault occurs in accessing memory operands.  
                      If the address in RCX points to a non-EPC page.  
                      If the address in RCX points to an invalid EPC page.

### 64-Bit Mode Exceptions

#GP(0)             If RCX is non-canonical form.  
                      If RCX points to a memory location not 8Byte-aligned.  
                      If the address in RCX points to a page belonging to a non-debug enclave.  
                      If the address in RCX points to a page which is not PT\_TCS, PT\_REG or PT\_VA.  
                      If the address in RCX points to a location inside TCS that is beyond SGX\_TCS\_LIMIT.

#PF(error code)    If a page fault occurs in accessing memory operands.  
                      If the address in RCX points to a non-EPC page.  
                      If the address in RCX points to an invalid EPC page.

## EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGWR (In)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

### Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

### EDBGWR Memory Parameter Semantics

EPCQW
Write access permitted by Enclave

The instruction faults if any of the following:

### EDBGWR Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

**Table 40-20. EDBGWR Return Value in RAX**

Error Code (see Table 40-4)	Description
No Error	EDBGWR successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGWR does not result in a #GP.

### Concurrency Restrictions

**Table 40-21. Base Concurrency Restrictions of EDBGWR**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGWR	Target [DS:RCX]	Shared	#GP	

**Table 40-22. Additional Concurrency Restrictions of EDBGWR**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGWR	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EDBGWR Operational Flow**

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP\_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )  
 THEN #GP(0); FI;

IF ( (TMP\_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing 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:RCX).MODIFIED is not 0) )  
 THEN  
     RFLAGS.ZF ← 1;  
     RAX ← SGX\_PAGE\_NOT\_DEBUGGABLE;  
     GOTO DONE;

FI;

(\* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field\*)  
 IF ( (EPCM(DS:RCX).PT = PT\_TCS) and ((DS:RCX) & FF8H ≠ offset\_of\_FLAGS & 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)	<p>If the address in RCS violates DS limit or access rights.</p> <p>If DS segment is unusable.</p> <p>If RCX points to a memory location not 4Byte-aligned.</p> <p>If the address in RCX points to a page belonging to a non-debug enclave.</p> <p>If the address in RCX points to a page which is not PT_TCS or PT_REG.</p> <p>If the address in RCX points to a location inside TCS that is not the FLAGS word.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If the address in RCX points to a non-EPC page.</p> <p>If the address in RCX points to an invalid EPC page.</p>

### 64-Bit Mode Exceptions

#GP(0)	<p>If RCX is non-canonical form.</p> <p>If RCX points to a memory location not 8Byte-aligned.</p> <p>If the address in RCX points to a page belonging to a non-debug enclave.</p> <p>If the address in RCX points to a page which is not PT_TCS or PT_REG.</p> <p>If the address in RCX points to a location inside TCS that is not the FLAGS word.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If the address in RCX points to a non-EPC page.</p> <p>If the address in RCX points to an invalid EPC page.</p>

## EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized enclave page.

### Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

### Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string comprising of “EEXTEND” || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

### EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

The instruction faults if any of the following:

### EEXTEND Faulting Conditions

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

### Concurrency Restrictions

**Table 40-23. Base Concurrency Restrictions of EEXTEND**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXTEND	Target [DS:RCX]	Shared	#GP	
	SECS [DS:RBX]	Concurrent		



Table 40-24. Additional Concurrency Restrictions of EEXTEND

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXTEND	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	

## Operation

### Temp Variables in EEXTEND Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFS ET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RBX does resolve to an EPC page)  
THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

(\* make sure no other Intel SGX instruction is accessing EPCM \*)  
IF (Other instructions accessing EPCM)  
THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)  
THEN #PF(DS:RCX); FI;

(\* make sure that DS:RCX (DST) is pointing to a PT\_REG or PT\_TCS \*)  
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 ATTRIBUTES.INIT \*)  
IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS) )  
THEN #GP(0); FI;

(\* Calculate enclave offset \*)

TMP\_ENCLAVEOFFSET ← EPCM(DS:RCX).ENCLAVEADDRESS - TMP\_SECS.BASEADDR;  
 TMP\_ENCLAVEOFFSET ← TMP\_ENCLAVEOFFSET + (DS:RCX & 0FFFH)

(\* Add EEXTEND message and offset to MRENCLAVE \*)

TMPUPDATEFIELD[63:0] ← 00444E4554584545H; // "EEXTEND"  
 TMPUPDATEFIELD[127:64] ← TMP\_ENCLAVEOFFSET;  
 TMPUPDATEFIELD[511:128] ← 0; // 48 bytes  
 TMP\_SECS.MRENCLAVE ← SHA256UPDATE(TMP\_SECS.MRENCLAVE, TMPUPDATEFIELD)  
 INC enclave's MRENCLAVE update counter;

(\*Add 256 bytes to MRENCLAVE, 64 byte at a time \*)

TMP\_SECS.MRENCLAVE ← SHA256UPDATE(TMP\_SECS.MRENCLAVE, DS:RCX[511:0]);  
 TMP\_SECS.MRENCLAVE ← SHA256UPDATE(TMP\_SECS.MRENCLAVE, DS:RCX[1023: 512]);  
 TMP\_SECS.MRENCLAVE ← SHA256UPDATE(TMP\_SECS.MRENCLAVE, DS:RCX[1535: 1024]);  
 TMP\_SECS.MRENCLAVE ← SHA256UPDATE(TMP\_SECS.MRENCLAVE, DS:RCX[2047: 1536]);  
 INC enclave's MRENCLAVE update counter by 4;

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) 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:

$$q1 = \text{floor}(\text{Signature}^2 / \text{Modulus});$$

$$q2 = \text{floor}((\text{Signature}^3 - q1 * \text{Signature} * \text{Modulus}) / \text{Modulus});$$

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

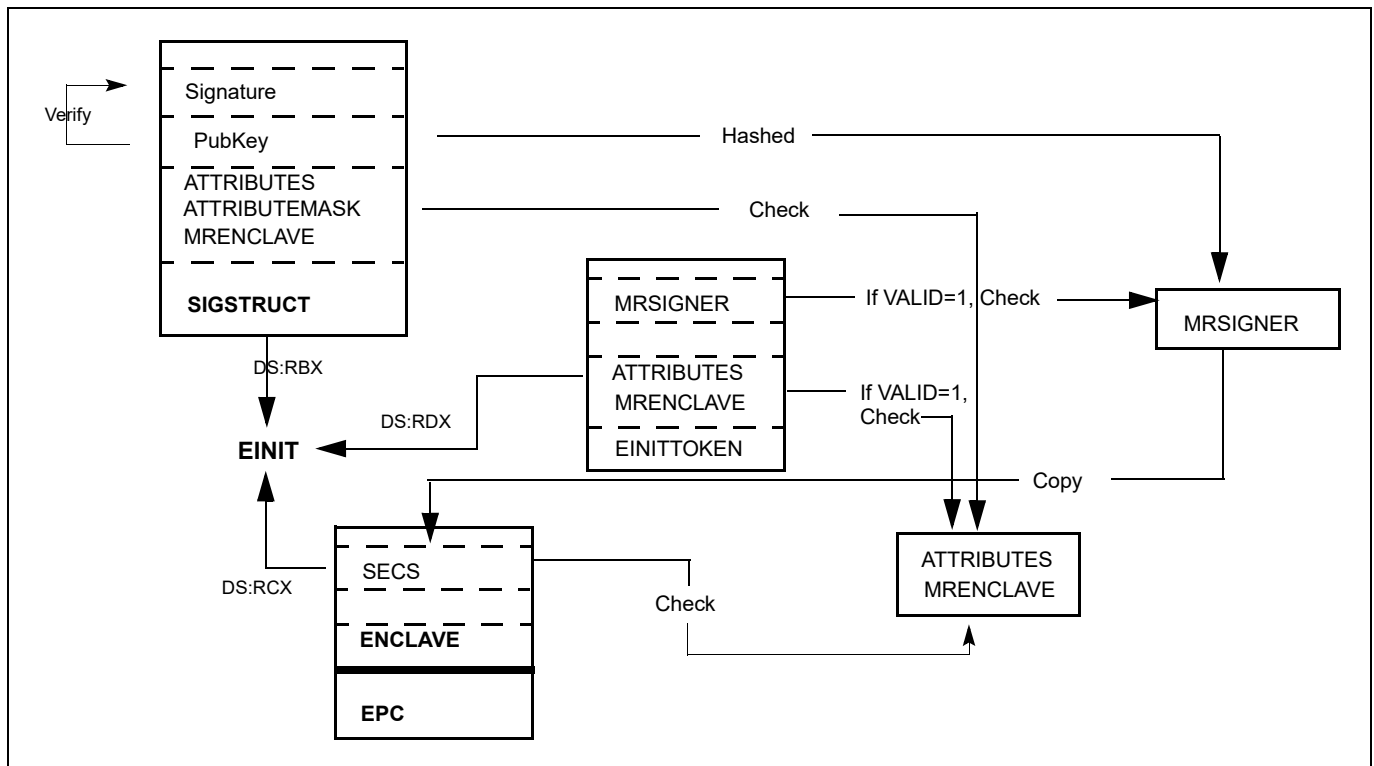


Figure 40-1. Relationships Between SECS, SIGSTRUCT and EINITTOKEN

**EINIT Memory Parameter Semantics**

SIGSTRUCT	SECS	EINITOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps, which can be seen in Figure 40-1:

Validates that SIGSTRUCT is signed using the enclosed public key.

Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.

Checks that no reserved bits are set to 1 in SIGSTRUCT.ATTRIBUTES and no reserved bits in SIGSTRUCT.ATTRIBUTESMASK are set to 0.

Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32\_SGX\_LEPUBKEYHASH.

Checks that SIGSTRUCT.ATTRIBUTES equals the result of logically and-ing SIGSTRUCT.ATTRIBUTESMASK with SECS.ATTRIBUTES.

If EINITOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32\_SGX\_LEPUBKEYHASH.

If EINITOKEN.VALID is 1, checks the validity of EINITOKEN.

If EINITOKEN.VALID is 1, checks that EINITOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITOKEN.VALID is 1 and EINITOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.

Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.

Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX\_UNMASKED\_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

**Table 40-25. EINIT Return Value in RAX**

Error Code (see Table 40-4)	Description
No Error	EINIT successful.
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value.
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask.
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement. If EINITOKEN contains an incorrect measurement.
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key.
SGX_INVALID_LICENSE	If license is invalid.
SGX_INVALID_CPUSVN	If license SVN is unsupported.
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation.

## Concurrency Restrictions

Table 40-26. Base Concurrency Restrictions of EINIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS [DS:RCX]	Shared	#GP	

Table 40-27. Additional Concurrency Restrictions of ENIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINIT	SECS [DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	

## Operation

## Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature <sup>3</sup> 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 ATTRIBUTES.INIT \*)

```
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS’s Initialized state))
    THEN #GP(0); FI;
```

(\* Calculate finalized version of MRENCLAVE \*)

(\* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest \*)

```
TMP_ENCLAVE ← SHA256FINAL( (DS:RCX).MRENCLAVE, enclave’s MRENCLAVE update count *512);
```

(\* Verify MRENCLAVE from SIGSTRUCT \*)

```
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
```

FI;

```
TMP_MRSIGNER ← SHA256(TMP_SIG.MODULUS)
```

```
(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
```

```
CONTROLLED_ATTRIBUTES ← 000000000000020H;
```

```
IF ( ( (DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES) ≠ 0) and (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH) )
```

```
    RFLAGS.ZF ← 1;
```

```
    RAX ← SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
```

```
IF ( (DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK) ≠ (TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK) )
```

```
    RFLAGS.ZF ← 1;
```

```
    RAX ← SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(*Verify SIGSTRUCT.MISCSELECT requirements are met *)
```

```
IF ( (DS:RCX.MISCSELECT & TMP_SIG.MISCMASK) ≠ (TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK) )
```

```
    THEN
```

```
        RFLAGS.ZF ← 1;
```

```
        RAX ← SGX_INVALID_ATTRIBUTE;
```

```
    GOTO EXIT
```

```
FI;
```

```
(* if EINITOKEN.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_EINITOKEN;
```

```
        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_EINITOKEN;
```

```
    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_EINITOKEN;
```

```
    GOTO EXIT;
```

```
FI;
```

```
(* EINIT token must be ≤ CR_CPUSVN *)
```

```
IF (TMP_TOKEN.CPUSVN > CR_CPUSVN)
```

```
    RFLAGS.ZF ← 1;
```

```
    RAX ← SGX_INVALID_CPUSVN;
```

## SGX INSTRUCTION REFERENCES

```
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)	<p>If a memory operand is not properly aligned.</p> <p>If another instruction is modifying the SECS.</p> <p>If the enclave is already initialized.</p> <p>If the SECS.MRENCLAVE is in use.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RCX does not resolve in an EPC page.</p> <p>If the memory address is not a valid, uninitialized SECS.</p>

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory operand is not properly aligned.</p> <p>If another instruction is modifying the SECS.</p> <p>If the enclave is already initialized.</p> <p>If the SECS.MRENCLAVE is in use.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RCX does not resolve in an EPC page.</p> <p>If the memory address is not a valid, uninitialized SECS.</p>

### ELDB/ELDU/ELDBC/ELBUC—Load an EPC Page and Mark its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.
EAX = 12H ENCLS[ELDBC]	IR	V/V	EAX[5]	This leaf function behaves like ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELBUC]	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/ELBUC 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;

## SGX INSTRUCTION REFERENCES

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

```
(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;
```

```
TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECS ← DS:RBX.SECONDS;
TMP_PCMD ← DS:RBX.PCMD;
```

```
(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;
```

```
(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            THEN
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason ← SGX_CONFLICT;
                        VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                        VMCS.Exit_qualification.error ← 0;
                        VMCS.Guest-physical_address ←
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address ← DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
                IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason ← SGX_CONFLICT;
                        VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_ERROR;
                        VMCS.Exit_qualification.error ← SGX_EPC_PAGE_CONFLICT;
                        VMCS.Guest-physical_address ←
                            << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address ← DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        RFLAGS.ZF ← 1;
                        RFLAGS.CF ← 0;
                        RAX ← SGX_EPC_PAGE_CONFLICT;
                        GOTO ERROR_EXIT;
                FI;
```

```

    FI;
FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot)
    THEN
        IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
            #GP(0);
            FI;
        ELSE (* ELDBC/ELDUC *)
            RFLAGS.ZF ← 1;
            RFLAGS.CF ← 0;
            RAX ← SGX_EPC_PAGE_CONFLICT;
            GOTO ERROR_EXIT;
        FI;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~OFFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0] ← DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0] ← 0;

TMP_HEADER.SECINFO ← SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD ← SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR ← DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
    (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) )
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Other instructions modifying SECS)
            THEN
                IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
                    #GP(0);
                    FI;
                ELSE (* ELDBC/ELDUC *)
                    RFLAGS.ZF ← 1;
                    RFLAGS.CF ← 0;
                    RAX ← SGX_EPC_PAGE_CONFLICT;
                    GOTO ERROR_EXIT;
                FI;
        FI;
FI;

```

## SGX INSTRUCTION REFERENCES

```
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_SRC_PGE[32767: 0];
TMP_VER ← DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} ← AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
  THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_MAC_COMPARE_FAIL;
    GOTO ERROR_EXIT;
FI;

(* Check version before committing *)
IF (DS:RDX ≠ 0)
  THEN #GP(0);
  ELSE
    DS:RDX ← TMP_VER;
FI;

(* Commit EPCM changes *)
EPCM(DS:RCX).PT ← TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX ← TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING ← TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED ← TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR ← TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA) )
  THEN
    EPCM(DS:RCX).BLOCKED ← 1;
  ELSE
    EPCM(DS:RCX).BLOCKED ← 0;
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
  << store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;

EPCM(DS:RCX).VALID ← 1;

RAX ← 0;
RFLAGS.ZF ← 0;
```

ERROR\_EXIT:  
RFLAGS.CF,PF,AF,OF,SF ← 0;

### Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the instruction's EPC resource is in use by others.</li> <li>If the instruction fails to verify MAC.</li> <li>If the version-array slot is in use.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand expected to be in EPC does not resolve to an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> <li>If the destination EPC page is already valid.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the instruction's EPC resource is in use by others.</li> <li>If the instruction fails to verify MAC.</li> <li>If the version-array slot is in use.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand expected to be in EPC does not resolve to an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> <li>If the destination EPC page is already valid.</li> </ul>

## EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

### EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

### EMODPR Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

**Table 40-31. EMODPR Return Value in RAX**

Error Code (see Table 40-4)	Description
No Error	EMODPR successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

### Concurrency Restrictions

**Table 40-32. Base Concurrency Restrictions of EMODPR**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPR	Target [DS:RCX]	Shared	#GP	



Table 40-33. Additional Concurrency Restrictions of EMODPR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

## Operation

### Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
THEN #PF(DS:RCX); FI;

SCRATCH\_SECINFO ← DS:RBX;

(\* Check for misconfigured SECINFO flags\*)  
IF ( (SCRATCH\_SECINFO reserved fields are not zero ) or  
(SCRATCH\_SECINFO.FLAGS.R is 0 and SCRATCH\_SECINFO.FLAGS.W is not 0) )  
THEN #GP(0); FI;

(\* Check concurrency with SGX1 or SGX2 instructions on the EPC page \*)  
IF (SGX1 or other SGX2 instructions accessing EPC page)  
THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0 )  
THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)  
IF (EPC page in use by another SGX2 instruction)  
THEN  
RFLAGS.ZF ← 1;  
RAX ← SGX\_EPC\_PAGE\_CONFLICT;  
GOTO DONE;

FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )  
THEN  
RFLAGS.ZF ← 1;  
RAX ← SGX\_PAGE\_NOT\_MODIFIABLE;

## SGX INSTRUCTION REFERENCES

```
GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR ← 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

### Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

### 64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page.

## EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

### EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

### EMODT Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

**Table 40-34. EMODT Return Value in RAX**

Error Code (see Table 40-4)	Description
No Error	EMODT successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.

### Concurrency Restrictions

**Table 40-35. Base Concurrency Restrictions of EMODT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR

**Table 40-36. Additional Concurrency Restrictions of EMODT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

**Operation**

**Temp Variables in EMODT Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
 THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)  
 THEN #PF(DS:RCX); FI;

SCRATCH\_SECINFO ← DS:RBX;

(\* Check for misconfigured SECINFO flags\*)

IF ( (SCRATCH\_SECINFO reserved fields are not zero ) or  
 !(SCRATCH\_SECINFO.FLAGS.PT is PT\_TCS or SCRATCH\_SECINFO.FLAGS.PT is PT\_TRIM) )  
 THEN #GP(0); FI;

(\* Check concurrency with SGX1 instructions on the EPC page \*)

IF (other SGX1 instructions accessing EPC page)  
 THEN  
     RFLAGS.ZF ← 1;  
     RAX ← SGX\_EPC\_PAGE\_CONFLICT;  
     GOTO DONE;

FI;

IF (EPCM(DS:RCX).VALID is 0)  
 THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)

IF (EPC page in use by another SGX2 instruction)  
 THEN  
     RFLAGS.ZF ← 1;  
     RAX ← SGX\_EPC\_PAGE\_CONFLICT;  
     GOTO DONE;

```

FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
      (EPCM(DS:RCX).PT is PT_TCS 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)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If another Intel SGX instruction is accessing the EPC page.</li> <li>If RBX is not set to PT_VA.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If the EPC page is valid.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If another Intel SGX instruction is accessing the EPC page.</li> <li>If RBX is not set to PT_VA.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If the EPC page is valid.</li> </ul>

## 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  $\leftarrow$  1;  
 RFLAGS.ZF  $\leftarrow$  0;  
 RAX  $\leftarrow$  SGX\_PG\_NONEPC;  
 goto DONE;  
 FI;

(\* Check the EPC page for concurrency \*)  
 IF (EPC page is being modified) THEN  
 RFLAGS.ZF = 1;  
 RFLAGS.CF = 0;  
 RAX = SGX\_EPC\_PAGE\_CONFLICT;  
 goto DONE;  
 FI;

(\* check page validity \*)  
 IF (EPCM(DS:RCX).VALID = 0) THEN  
 RFLAGS.CF = 1;

## SGX INSTRUCTION REFERENCES

```
RFLAGS.ZF = 0;
RAX = SGX_PG_INVLD;
goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO ← DS:RBX;
TMP_RDINFO.STATUS ← 0;
TMP_RDINFO.FLAGS ← 0;
TMP_RDINFO.ENCLAVECONTEXT ← 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
TMP_RDINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR ← EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE ← EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED ← EPCM(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TRIM)
    ) 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;

## SGX INSTRUCTION REFERENCES

```
IF ( (EPCM(DS:RCX).PT = PT_VA) OR
      ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
  THEN
    EPCM(DS:RCX).VALID ← 0;
    GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
  THEN
    IF (DS:RCX has an EPC page associated with it)
      THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_CHILD_PRESENT;
        GOTO ERROR_EXIT;

    FI;
    (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
    IF (<<in VMX non-root operation>> AND
        <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
        (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
      THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_CHILD_PRESENT
        GOTO ERROR_EXIT

    FI;
    EPCM(DS:RCX).VALID ← 0;
    GOTO DONE;
FI;

IF (Other threads active using SECS)
  THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_ENCLAVE_ACT;
    GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) )
  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;
```

```
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.
- #PF(error code)    If another thread is concurrently using the tracking facility on this SECS.  
                          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.
- #PF(error code)    If the memory operand is not properly aligned.  
                          If the memory operand expected to be in EPC does not resolve to an EPC page.  
                          If a page fault occurs in access memory operand.

**64-Bit Mode Exceptions**

- #GP(0)                If a memory address is in a non-canonical form.  
                          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_SRCPAGE	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\_SRCPAGE ← DS:RBX.SRCPGE;  
 (\* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO \*)  
 TMP\_PCMD ← DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)  
 THEN #GP(0); FI;

■ IF ( (DS:TMP\_PCMD is not 128Byte Aligned) or (DS:TMP\_SRCPAGE is not 4KByte Aligned) )  
 THEN #GP(0); FI;

(\* Check for concurrent Intel SGX instruction access to the page \*)  
 IF (Other Intel SGX instruction is accessing page)  
 THEN  
     IF (<<VMX non-root operation>> AND <<ENABLE\_EPC\_VIRTUALIZATION\_EXTENSIONS>>)  
     THEN  
         VMCS.Exit\_reason ← SGX\_CONFLICT;  
         VMCS.Exit\_qualification.code ← EPC\_PAGE\_CONFLICT\_EXCEPTION;  
         VMCS.Exit\_qualification.error ← 0;  
         VMCS.Guest-physical\_address ← << translation of DS:RCX produced by paging >>;



```

        VMCS.Guest-linear_address ← DS:RCX;
        Deliver VMEXIT;
    ELSE
        #GP(0);
FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~OFFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) )
    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)	<ul style="list-style-type: none"> <li>If a memory operand effective address is outside the DS segment limit.</li> <li>If a memory operand is not properly aligned.</li> <li>If the EPC page and VASLOT resolve to the same EPC page.</li> <li>If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.</li> <li>If the tracking resource is in use.</li> <li>If the EPC page or the version array page is invalid.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> </ul>

### 64-Bit Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If a memory operand is non-canonical form.</li> <li>If a memory operand is not properly aligned.</li> <li>If the EPC page and VASLOT resolve to the same EPC page.</li> <li>If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.</li> <li>If the tracking resource is in use.</li> <li>If the EPC page or the version array page is invalid.</li> <li>If the parameters fail consistency checks.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory operands.</li> <li>If a memory operand is not an EPC page.</li> <li>If one of the EPC memory operands has incorrect page type.</li> </ul>

## 40.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

## EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

### Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

### Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

### EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

### EACCEPT Faulting Conditions

The operands are not properly aligned.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	Page type is PT_REG and MODIFIED bit is 0.
SECINFO contains an invalid request.	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
If security attributes of the SECINFO page make the page inaccessible.	

The error codes are:

**Table 40-54. EACCEPT Return Value in RAX**

Error Code (see Table 40-4)	Description
No Error	EACCEPT successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page.

Concurrency Restrictions

Table 40-55. Base Concurrency Restrictions of EACCEPT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target [DS:RCX]	Shared	#GP	
	SECINFO [DS:RBX]	Concurrent		

Table 40-56. Additional Concurrency Restrictions of EACCEPT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

IF (DS:RBX is not 64Byte Aligned)  
 THEN #GP(0); FI;

IF (DS:RBX is not within CR\_ELRANGE)  
 THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)  
 THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or  
 (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or  
 (EPCM(DS:RBX &~FFFH).PT ≠ PT\_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
 (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)) )  
 THEN #PF(DS:RBX); FI;

(\* Copy 64 bytes of contents \*)  
 SCRATCH\_SECINFO ← DS:RBX;

(\* Check for misconfigured SECINFO flags\*)  
 IF (SCRATCH\_SECINFO reserved fields are not zero )  
 THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)  
 THEN #GP(0); FI;

```
IF (DS:RCX is not within CR_ELRANGE)
  THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
  THEN #PF(DS:RCX); FI;
```

(\* Check that the combination of requested PT, PENDING and MODIFIED is legal \*)

```
IF (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 misconfigured SECINFO flags\*)

IF ( (SCRATCH\_SECINFO reserved fields are not zero ) or (SCRATCH\_SECINFO.FLAGS.R=0) AND(SCRATCH\_SECINFO.FLAGS.W≠0) or  
 (SCRATCH\_SECINFO.FLAGS.PT is not PT\_REG) )  
 THEN #GP(0); FI;

(\* Check security attributes of the source EPC page \*)

IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING ≠ 0) or (EPCM(DS:RDX).MODIFIED ≠ 0) or  
 (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RDX).PT ≠ PT\_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or  
 (EPCM(DS:RDX).ENCLAVEADDRESS ≠ DS:RDX))  
 THEN #PF(DS:RDX); FI;

(\* Check security attributes of the destination EPC page \*)

IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or  
 (EPCM(DS: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 4Kbytes 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 pending on the instruction boundary immediately after EENTER (see Section 42.2.2).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):

- All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED\_CTR1 and FIXED\_CTR2.
- PEBS is suppressed.
- AnyThread counting on other threads is demoted to MyThread mode and IA32\_PERF\_GLOBAL\_STATUS[60] on that thread is set
- If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32\_PERF\_GLOBAL\_STATUS[60] and IA32\_PERF\_GLOBAL\_STATUS[63].

Concurrency Restrictions

Table 40-60. Base Concurrency Restrictions of EENTER

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EENTER	TCS [DS:RBX]	Shared	#GP	

Table 40-61. Additional Concurrency Restrictions of EENTER

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EENTER	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

(\* Make sure DS is usable, expand up \*)

IF (TMP\_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1) ) )  
 THEN #GP(0); FI;

(\* Check that CS, SS, DS, ES.base is 0 \*)

IF (TMP\_MODE64 = 0)  
 THEN  
 IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;  
 IF(ES usable and ES.base ≠ 0) #GP(0); FI;  
 IF(SS usable and SS.base ≠ 0) #GP(0); FI;  
 IF(SS usable and SS.B = 0) #GP(0); FI;

```

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 & FFFFFFFF7FFF) ≠ 0 )
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCRO) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one 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_ELRRANGE ← (TMPSECS.BASEADDR, TMP_SECS.SIZE);

```

(\* Save state for possible AEXs \*)

```

CR_TCS_PA ← Physical_Address (DS:RBX);
CR_TCS_LA ← RBX;

```

## SGX INSTRUCTION REFERENCES

CR\_TCS\_LA.AEP ← RCX;

(\* Save the hidden portions of FS and GS \*)

CR\_SAVE\_FS\_selector ← FS.selector;

CR\_SAVE\_FS\_base ← FS.base;

CR\_SAVE\_FS\_limit ← FS.limit;

CR\_SAVE\_FS\_access\_rights ← FS.access\_rights;

CR\_SAVE\_GS\_selector ← GS.selector;

CR\_SAVE\_GS\_base ← GS.base;

CR\_SAVE\_GS\_limit ← GS.limit;

CR\_SAVE\_GS\_access\_rights ← GS.access\_rights;

(\* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM\*)

IF (CR4.OSXSAVE = 1)

    CR\_SAVE\_XCRO ← XCRO;

    XCRO ← TMP\_SECS.ATTRIBUTES.XFRM;

FI;

RCX ← RIP;

RIP ← TMP\_TARGET;

RAX ← (DS:RBX).CSSA;

(\* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT \*)

DS:TMP\_SSA.U\_RSP ← RSP;

DS:TMP\_SSA.U\_RBP ← RBP;

(\* Do the FS/GS swap \*)

FS.base ← TMP\_FSBASE;

FS.limit ← DS:RBX.FSLIMIT;

FS.type ← 0001b;

FS.W ← DS.W;

FS.S ← 1;

FS.DPL ← DS.DPL;

FS.G ← 1;

FS.B ← 1;

FS.P ← 1;

FS.AVL ← DS.AVL;

FS.L ← DS.L;

FS.unusable ← 0;

FS.selector ← 0BH;

GS.base ← TMP\_GSBASE;

GS.limit ← DS:RBX.GSLIMIT;

GS.type ← 0001b;

GS.W ← DS.W;

GS.S ← 1;

GS.DPL ← DS.DPL;

GS.G ← 1;

GS.B ← 1;

GS.P ← 1;

GS.AVL ← DS.AVL;

GS.L ← DS.L;

GS.unusable ← 0;

GS.selector ← 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)	<ul style="list-style-type: none"> <li>If DS:RBX is not page aligned.</li> <li>If the enclave is not initialized.</li> <li>If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.</li> <li>If the thread is not in the INACTIVE state.</li> <li>If CS, DS, ES or SS bases are not all zero.</li> <li>If executed in enclave mode.</li> <li>If any reserved field in the TCS FLAG is set.</li> <li>If the target address is not within the CS segment.</li> <li>If CR4.OSFXSR = 0.</li> <li>If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.</li> <li>If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCRO.</li> </ul>
#PF(error code)	<ul style="list-style-type: none"> <li>If a page fault occurs in accessing memory.</li> <li>If DS:RBX does not point to a valid TCS.</li> <li>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.</li> </ul>

### 64-Bit Mode Exceptions

- #GP(0)
  - If DS:RBX is not page aligned.
  - If the enclave is not initialized.
  - If the thread is not in the INACTIVE state.
  - If CS, DS, ES or SS bases are not all zero.
  - If executed in enclave mode.
  - If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
  - If the target address is not canonical.
  - If CR4.OSFXSR = 0.
  - If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
  - If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
- #PF(error code)
  - If a page fault occurs in accessing memory operands.
  - If DS:RBX does not point to a valid TCS.
  - If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT\_REG EPC page.

## EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

### Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (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 fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

### Concurrency Restrictions

**Table 40-62. Base Concurrency Restrictions of EEXIT**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXIT		Concurrent		

**Table 40-63. Additional Concurrency Restrictions of EEXIT**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXIT		Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

```
IF (TMP_MODE64 = 1)
    THEN
        IF (RBX is not canonical) THEN #GP(0); FI;
    ELSE
        IF (RBX > CS limit) THEN #GP(0); FI;
FI;
```

TMP\_RIP ← CRIP;  
RIP ← RBX;

(\* Return current AEP in RCX \*)  
RCX ← CR\_TCS\_PA.AEP;

(\* Do the FS/GS swap \*)  
FS.selector ← CR\_SAVE\_FS.selector;  
FS.base ← CR\_SAVE\_FS.base;  
FS.limit ← CR\_SAVE\_FS.limit;  
FS.access\_rights ← CR\_SAVE\_FS.access\_rights;  
GS.selector ← CR\_SAVE\_GS.selector;  
GS.base ← CR\_SAVE\_GS.base;  
GS.limit ← CR\_SAVE\_GS.limit;  
GS.access\_rights ← CR\_SAVE\_GS.access\_rights;

(\* Restore XCRO if needed \*)  
IF (CR4.OSXSAVE = 1)  
 XCRO ← CR\_SAVE\_\_XCRO;  
FI;

Unsuppress\_all\_code\_breakpoints\_that\_are\_outside\_ELRANGE;

```
IF (CR_DBGOPTIN = 0)
    THEN
        UnSuppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        Restore suppressed breakpoint matches;
        RFLAGS.TF ← CR_SAVE_TF;
        UnSuppress_montior_trap_flag;
        UnSuppress_LBR_Generation;
        UnSuppress_performance_monitoring_activity;
        Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;
```

```
IF RFLAGS.TF = 1
    THEN Pend Single-Step #DB at the end of EEXIT;
FI;
```

IF the “monitor trap flag” VM-execution control is set  
THEN pend a MTF VM exit at the end of EEXIT;  
FI;

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.

### EGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 40-64.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX\_INVALID\_SVN, SGX\_INVALID\_ATTRIBUTE, SGX\_INVALID\_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

### Requesting Keys

The KEYREQUEST structure (see Section 37.17.1) 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.



**Table 40-64. Key Derivation**

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PROPID	ISVEXT PROPID	ISVFAM ILYID	MRENCLAVE	MRSIGNER	CONFIG ID	CONFIGS VN	RAND
Source	Key Dependent Constant	Y← SECS.ATTRIBUTES and SECS.MISCSELECT;	CR_SGX OWNER EPOCH	Y← CPUSVN Register;	R← Req.ISV SVN;	SECS. ISVID	SECS.IS VEXTPROPID	SECS.IS VFAMILYID	SECS. MRENCLAVE	SECS. MRSIGNER	SECS.CO NFIGID	SECS.CO NFIGSVN	Req. KEYID
		R←AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT;		R← Req.CPU SVN;									
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	No	No	Yes	No	No	Request
Report	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	Request
Seal	Yes	Request	Yes	Request	Request	Request	Request	Request	Request	Request	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	No	No	Yes	No	No	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Request	Request	Request	No	Yes	Request	Request	Yes

Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN\_KEY be set and SECS.MRSIGNER equal IA32\_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcoded PKCS padding constant (352 byte string):

HARDCODED\_PKCS1\_5\_PADDING[15:0] ← 0100H;

HARDCODED\_PKCS1\_5\_PADDING[2655:16] ← SignExtend330Byte(-1); // 330 bytes of 0FFH

HARDCODED\_PKCS1\_5\_PADDING[2815:2656] ← 2004000501020403650148866009060D30313000H;

The error codes are:

**Table 40-65. EGETKEY Return Value in RAX**

Error Code (see Table 40-4)	Value	Description
No Error	0	EGETKEY successful.
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.
SGX_INVALID_ISVSVN		If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave's corresponding SVN.
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value.

**Concurrency Restrictions**

**Table 40-66. Base Concurrency Restrictions of EGETKEY**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		
	OUTPUTDATA [DS:RCX]	Concurrent		

**Table 40-67. Additional Concurrency Restrictions of EGETKEY**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EGETKEY Operational Flow**

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_ISVEXTPRODID		16 bytes	Temp Space for ISVEXTPRODID.
TMP_ISVPRODID		2 bytes	Temp Space for ISVPRODID.
TMP_ISVFAMILYID		16 bytes	Temp Space for ISVFAMILYID.
TMP_CONFIGID		64 bytes	Temp Space for CONFIGID.
TMP_CONFIGSVN		2 bytes	Temp Space for CONFIGSVN.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

(\* Make sure KEYREQUEST is properly aligned and inside the current enclave \*)

IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is within CR\_ELRANGE) )  
 THEN #GP(0); FI;

(\* Make sure DS:RBX is an EPC address and the EPC page is valid \*)

IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )  
 THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)

THEN #PF(DS:RBX); FI;

(\* Check page parameters for correctness \*)

IF ( (EPCM(DS:RBX).PT ≠ PT\_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or (EPCM(DS:RBX).PENDING = 1) or  
 (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )  
 THEN #PF(DS:RBX);

FI;

(\* Make sure OUTPUTDATA is properly aligned and inside the current enclave \*)

IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR\_ELRANGE) )  
 THEN #GP(0); FI;

(\* Make sure DS:RCX is an EPC address and the EPC page is valid \*)

IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )

```

THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
  THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
  (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~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?*)

```

## SGX INSTRUCTION REFERENCES

```
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;

```

## SGX INSTRUCTION REFERENCES

```
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 misconfigured SECINFO flags\*)  
IF (SCRATCH\_SECINFO reserved fields are not zero )  
THEN #GP(0); FI;

(\* Check security attributes of the EPC page \*)  
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or (EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT\_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) )  
THEN #PF(DS:RCX); FI;

(\* Check the EPC page for concurrency \*)  
IF (EPC page in use by another SGX2 instruction)  
THEN #GP(0); FI;

(\* Re-Check security attributes of the EPC page \*)  
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT\_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR\_ACTIVE\_SECS) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))  
THEN #PF(DS:RCX); FI;

**■** (\* Check for misconfigured SECINFO flags\*)  
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH\_SECINFO.FLAGS.R = 0) and (SCRATCH\_SECINFO.FLAGS.W ≠ 0) )  
THEN #GP(0); FI;

(\* Update EPCM permissions \*)

EPCM(DS:RCX).R ← EPCM(DS:RCX).R | SCRATCH\_SECINFO.FLAGS.R;

EPCM(DS:RCX).W ← EPCM(DS:RCX).W | SCRATCH\_SECINFO.FLAGS.W;

EPCM(DS:RCX).X ← EPCM(DS:RCX).X | SCRATCH\_SECINFO.FLAGS.X;

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)                    If 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) )

## SGX INSTRUCTION REFERENCES

```
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 #PF(DS:RCX); FI;
```

```
IF (EPCM(DS:RCX).VALID = 0)  
    THEN #PF(DS:RCX); FI;
```

```
IF (EPCM(DS:RCX).BLOCKED = 1)  
    THEN #PF(DS:RCX); FI;
```

```
(* Check page parameters for correctness *)  
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or  
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~OFFFH) ) 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 & ~OFFFH) ) 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 pending on the instruction boundary immediately after EENTER (see Section 42.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.



- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED\_CTR1 and FIXED\_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32\_PERF\_GLOBAL\_STATUS[60] on that thread is set.
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32\_PERF\_GLOBAL\_STATUS[60] and IA32\_PERF\_GLOBAL\_STATUS[63].

## Concurrency Restrictions

**Table 40-72. Base Concurrency Restrictions of ERESUME**

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERESUME	TCS [DS:RBX]	Shared	#GP	

**Table 40-73. Additional Concurrency Restrictions of ERESUME**

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERESUME	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

## Operation

### Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for GS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.

TMP\_MODE64 ← ((IA32\_EFER.LMA = 1) && (CS.L = 1));

(\* Make sure DS is usable, expand up \*)

IF (TMP\_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))  
 THEN #GP(0); FI;

## SGX INSTRUCTION REFERENCES

(\* Check that CS, SS, DS, ES.base is 0 \*)

```
IF (TMP_MODE64 = 0)
  THEN
    IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
    IF(ES usable and ES.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.base ≠ 0) #GP(0); FI;
    IF(SS usable and SS.B = 0) #GP(0); FI;
FI;
```

IF (DS:RBX is not 4KByte Aligned)

```
  THEN #GP(0); FI;
```

IF (DS:RBX does not resolve within an EPC)

```
  THEN #PF(DS:RBX); FI;
```

(\* Check AEP is canonical\*)

```
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
  THEN #GP(0); FI;
```

(\* Check concurrency of TCS operation\*)

```
IF (Other Intel SGX instructions is operating on TCS)
  THEN #GP(0); FI;
```

(\* TCS verification \*)

```
IF (EPCM(DS:RBX).VALID = 0)
  THEN #PF(DS:RBX); FI;
```

IF (EPCM(DS:RBX).BLOCKED = 1)

```
  THEN #PF(DS:RBX); FI;
```

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))

```
  THEN #PF(DS:RBX); FI;
```

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT\_TCS) )

```
  THEN #PF(DS:RBX); FI;
```

IF ( (DS:RBX).OSSA is not 4KByte Aligned)

```
  THEN #GP(0); FI;
```

(\* Check proposed FS and GS \*)

```
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
  THEN #GP(0); FI;
```

(\* Get the SECS for the enclave in which the TCS resides \*)

```
TMP_SECS ← Address of SECS for TCS;
```

(\* Make sure that the FLAGS field in the TCS does not have any reserved bits set \*)

```
IF ( ( (DS:RBX).FLAGS & FFFFFFFF00000000H) ≠ 0)
  THEN #GP(0); FI;
```

(\* SECS must exist and enclave must have previously been EINITted \*)

```
IF (the enclave is not already initialized)
  THEN #GP(0); FI;
```

(\* make sure the logical processor's operating mode matches the enclave \*)

```
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;
```

```
IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;
```

(\* Check for legal values of SECS.ATTRIBUTES.XFRM \*)

```
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & 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:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
```

```
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
```

```
    THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
    CR_XSAVE_PAGE_n ← Physical_Address(DS:TMP_SSA_PAGE);
```

```
ENDFOR
```

(\* Compute address of GPR area\*)

```
TMP_GPR ← TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

Check that DS:TMP\_SSA\_PAGE is read/write accessible;

If a fault occurs, release locks, abort and deliver that fault;

```
IF (DS:TMP_GPR does not resolve to EPC page)
```

```
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).VALID = 0)
```

```
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
```

```
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
```

```
    THEN #PF(DS:TMP_GPR); FI;
```

## SGX INSTRUCTION REFERENCES

```
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
(EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
(EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
THEN #PF(DS:TMP_GPR); FI;
```

```
IF (TMP_MODE64 = 0)
THEN
    IF (TMP_GPR + (GPR_SIZE - 1) is not in DS segment) THEN #GP(0); FI;
FI;
```

```
CR_GPR_PA ← Physical_Address (DS: TMP_GPR);
```

```
TMP_TARGET ← (DS:TMP_GPR).RIP;
IF (TMP_MODE64 = 1)
THEN
    IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
ELSE
    IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;
```

(\* Check proposed FS/GS segments fall within DS \*)

```
IF (TMP_MODE64 = 0)
THEN
    TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_FSLIMIT ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
    TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    TMP_GSLIMIT ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
    (* if FS wrap-around, make sure DS has no holes*)
    IF (TMP_FSLIMIT < TMP_FSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
    (* if GS wrap-around, make sure DS has no holes*)
    IF (TMP_GSLIMIT < TMP_GSBASE)
    THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
    ELSE
        IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
    FI;
ELSE
    TMP_FSBASE ← DS:TMP_GPR.FSBASE;
    TMP_GSBASE ← DS:TMP_GPR.GSBASE;
    IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
    THEN #GP(0); FI;
FI;
```

(\* Ensure the enclave is not already active and this thread is the only one using the TCS\*)

```
IF (DS:RBX.STATE = ACTIVE)
THEN #GP(0); FI;
```

(\* 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_ELRRANGE ← (TMP_SECS.BASEADDR, TMP_SECS.SIZE);
```

```
(* Save state for possible AEXs *)
```

```
CR_TCS_PA ← Physical_Address (DS:RBX);
```

```
CR_TCS_LA ← RBX;
```

```
CR_TCS_LA.AEP ← RCX;
```

```
(* Save the hidden portions of FS and GS *)
```

```
CR_SAVE_FS_selector ← FS.selector;
```

```
CR_SAVE_FS_base ← FS.base;
```

```
CR_SAVE_FS_limit ← FS.limit;
```

```
CR_SAVE_FS_access_rights ← FS.access_rights;
```

```
CR_SAVE_GS_selector ← GS.selector;
```

```
CR_SAVE_GS_base ← GS.base;
```

```
CR_SAVE_GS_limit ← GS.limit;
```

```
CR_SAVE_GS_access_rights ← GS.access_rights;
```

```
RIP ← TMP_TARGET;
```

```
Restore_GPRs from DS:TMP_GPR;
```

```
(*Restore the RFLAGS values from SSA*)
```

```
RFLAGS.CF ← DS:TMP_GPR.RFLAGS.CF;
```

```
RFLAGS.PF ← DS:TMP_GPR.RFLAGS.PF;
```

```
RFLAGS.AF ← DS:TMP_GPR.RFLAGS.AF;
```

```
RFLAGS.ZF ← DS:TMP_GPR.RFLAGS.ZF;
```

```
RFLAGS.SF ← DS:TMP_GPR.RFLAGS.SF;
```

```
RFLAGS.DF ← DS:TMP_GPR.RFLAGS.DF;
```

```
RFLAGS.OF ← DS:TMP_GPR.RFLAGS.OF;
```

```
RFLAGS.NT ← DS:TMP_GPR.RFLAGS.NT;
```

```
RFLAGS.AC ← DS:TMP_GPR.RFLAGS.AC;
```

```
RFLAGS.ID ← DS:TMP_GPR.RFLAGS.ID;
```

```
RFLAGS.RF ← DS:TMP_GPR.RFLAGS.RF;
```

```
RFLAGS.VM ← 0;
```

```
IF (RFLAGS.IOPL = 3)
```

```
  THEN RFLAGS.IF ← DS:TMP_GPR.RFLAGS.IF; FI;
```

```
IF (TCS.FLAGS.OPTIN = 0)
```

```
  THEN RFLAGS.TF ← 0; FI;
```

```
(* If XSAVE is enabled, save XCRO and replace it with SECS.ATTRIBUTES.XFRM*)
```

```
IF (CR4.OSXSAVE = 1)
```

```
  CR_SAVE_XCRO ← XCRO;
```

```
  XCRO ← TMP_SECS.ATTRIBUTES.XFRM;
```

## SGX INSTRUCTION REFERENCES

FI;

(\* Pop the SSA stack\*)  
(DS:RBX).CSSA  $\leftarrow$  (DS:RBX).CSSA - 1;

(\* Do the FS/GS swap \*)  
FS.base  $\leftarrow$  TMP\_FSBASE;  
FS.limit  $\leftarrow$  DS:RBX.FSLIMIT;  
FS.type  $\leftarrow$  0001b;  
FS.W  $\leftarrow$  DS.W;  
FS.S  $\leftarrow$  1;  
FS.DPL  $\leftarrow$  DS.DPL;  
FS.G  $\leftarrow$  1;  
FS.B  $\leftarrow$  1;  
FS.P  $\leftarrow$  1;  
FS.AVL  $\leftarrow$  DS.AVL;  
FS.L  $\leftarrow$  DS.L;  
FS.unusable  $\leftarrow$  0;  
FS.selector  $\leftarrow$  0BH;

GS.base  $\leftarrow$  TMP\_GSBASE;  
GS.limit  $\leftarrow$  DS:RBX.GSLIMIT;  
GS.type  $\leftarrow$  0001b;  
GS.W  $\leftarrow$  DS.W;  
GS.S  $\leftarrow$  1;  
GS.DPL  $\leftarrow$  DS.DPL;  
GS.G  $\leftarrow$  1;  
GS.B  $\leftarrow$  1;  
GS.P  $\leftarrow$  1;  
GS.AVL  $\leftarrow$  DS.AVL;  
GS.L  $\leftarrow$  DS.L;  
GS.unusable  $\leftarrow$  0;  
GS.selector  $\leftarrow$  0BH;

CR\_DBGOPTIN  $\leftarrow$  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  $\leftarrow$  RFLAGS.TF;  
    RFLAGS.TF  $\leftarrow$  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)	<p>If DS:RBX is not page aligned.</p> <p>If the enclave is not initialized.</p> <p>If the thread is not in the INACTIVE state.</p> <p>If CS, DS, ES or SS bases are not all zero.</p> <p>If executed in enclave mode.</p> <p>If part or all of the FS or GS segment specified by TCS is outside the DS segment.</p> <p>If any reserved field in the TCS FLAG is set.</p> <p>If the target address is not within the CS segment.</p> <p>If CR4.OSFXSR = 0.</p> <p>If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.</p> <p>If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory.</p> <p>If DS:RBX does not point to a valid TCS.</p> <p>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.</p>

**64-Bit Mode Exceptions**

#GP(0)	<p>If DS:RBX is not page aligned.</p> <p>If the enclave is not initialized.</p> <p>If the thread is not in the INACTIVE state.</p> <p>If CS, DS, ES or SS bases are not all zero.</p> <p>If executed in enclave mode.</p> <p>If part or all of the FS or GS segment specified by TCS is outside the DS segment.</p> <p>If any reserved field in the TCS FLAG is set.</p> <p>If the target address is not canonical.</p> <p>If CR4.OSFXSR = 0.</p> <p>If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.</p> <p>If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If DS:RBX does not point to a valid TCS.</p> <p>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.</p>

## 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 EDECVIRTUALCHILD**

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
EDECVIRTUALCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EDECVIRTUALCHILD Operational Flow**

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_VIRTUALCHILDCNT	Integer	64	Number of virtual child pages.

**EDECVIRTUALCHILD Return Value in RAX**

Error	Value	Description
No Error	0	EDECVIRTUALCHILD 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)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If DS segment is unusable.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

**64-Bit Mode Exceptions**

- #GP(0)            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 EINCVRTCHILD**

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
EINCVRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

**Operation**

**Temp Variables in EINCVRTCHILD 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.

**EINCVRTCHILD Return Value in RAX**

Error	Value	Description
No Error	0	EINCVRTCHILD 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)	<p>If a memory operand effective address is outside the DS segment limit.</p> <p>If DS segment is unusable.</p> <p>If a memory operand is not properly aligned.</p> <p>RBX does not refer to an enclave page associated with SECS referenced in RCX.</p>
#PF(error code)	<p>If a page fault occurs in accessing memory operands.</p> <p>If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).</p> <p>If RCX does not refer to an SECS page.</p>

**64-Bit Mode Exceptions**

- #GP(0)            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);  
 FI;

(\* check EPC page is an SECS page \*)

```
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN
  #PF(DS:RCX, PFEC.SGX);
FI;
```

```
(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)
SECS(DS:RCX).ENCLAVECONTEXT ← TMP_CONTEXT;
```

```
RAX ← 0;
RFLAGS.ZF ← 0;
```

```
DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

### Flags Affected

ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared. CF, PF, AF, OF and SF are cleared.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit. If DS segment is unusable. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

### 64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form. If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.



### 23. Updates to Chapter 42, Volume 3D

Change bars show changes to Chapter 42 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: New information added; Section 42.4 "Consideration of the INT1 and INT3 Instructions" and sub-sections.

## CHAPTER 42

# ENCLAVE CODE DEBUG AND PROFILING

---

Intel® SGX is architected to provide protection for production enclaves and permit enclave code developers to use an SGX-aware debugger to effectively debug a non-production enclave (debug enclave). Intel SGX also allows a non-SGX-aware debugger to debug non-enclave portions of the application without getting confused by enclave instructions.

## 42.1 CONFIGURATION AND CONTROLS

### 42.1.1 Debug Enclave vs. Production Enclave

The SECS of each enclave provides a bit, SECS.ATTRIBUTES.DEBUG, indicating whether the enclave is a debug enclave (if set) or a production enclave (if 0). If this bit is set, software outside the enclave can use EDBGWR/EDBGWR to access the EPC memory of the enclave. The value of DEBUG is not included in the measurement of the enclave and therefore doesn't require an alternate SIGSTRUCT to be generated to debug the enclave.

The ATTRIBUTES field in the SECS is reported in the enclave's attestation, and is included in the key derivation. Enclave secrets that were protected by the enclave using Intel SGX keys when it ran as a production enclave will not be accessible by the debug enclave. A debugger needs to be aware that special debug content might be required for a debug enclave to run in a meaningful way.

EPC memory belonging to a debug enclave can be accessed via the EDBGWR/EDBGWR leaf functions (see Section 40.4), while that belonging to a non-debug enclave cannot be accessed by these leaf functions.

### 42.1.2 Tool-Chain Opt-in

The TCS.FLAGS.DBGOPTIN bit controls interactions of certain debug and profiling features with enclaves, including code/data breakpoints, TF, RF, monitor trap flag, BTF, LBRs, BTM, BTS, Intel Processor Trace, and performance monitoring. This bit is forced to zero when EPC pages are added via EADD. A debugger can set this bit via EDBGWR to the TCS of a debug enclave.

An enclave entry through a TCS with the TCS.FLAGS.DBGOPTIN set to 0 is called an **opt-out** entry. Conversely, an enclave entry through a TCS with TCS.FLAGS.DBGOPTIN set to 1 is called an **opt-in** entry.

## 42.2 SINGLE STEP DEBUG

### 42.2.1 Single Stepping ENCLS Instruction Leafs

If the RFLAGS.TF bit is set at the beginning of ENCLS, then a single-step debug exception is pending as a trap-class exception on the instruction boundary immediately after the ENCLS instruction. Additionally, if the instruction is executed in VMX non-root operation and the "monitor trap flag" VM-execution control is 1, an MTF VM exit is pending on the instruction boundary immediately after the instruction if the instruction does not fault.

### 42.2.2 Single Stepping ENCLU Instruction Leafs

The interactions of the unprivileged Intel SGX instruction ENCLU are leaf dependent.

An enclave entry via EENTER/ERESUME leaf functions of the ENCLU, in certain cases, may mask the RFLAGS.TF bit, and mask the setting of the "monitor trap flag" VM-execution control. In such situations, an exit from the enclave, either via the EEXIT leaf function or via an AEX unmask the RFLAGS.TF bit and the "monitor trap flag" VM-execu-

tion control. The details of this masking/unmasking and the pending of single stepping events across EENTER/ERESUME/EEXIT/AEX are covered in detail in Section 42.2.3.

If the EFLAGS.TF bit is set at the beginning of EREPORT or EGETKEY leafs, and if the EFLAGS.TF is not masked by the preceding enclave entry, then a single-step debug exception is pending on the instruction boundary immediately after the ENCLU instruction. Additionally, if the instruction is executed in VMX non-root operation and the “monitor trap flag” VM-execution control is 1, and if the monitor trap flag is not masked by the preceding enclave entry, then an MTF VM exit is pending on the instruction boundary immediately after the instruction.

If the instruction under consideration results in a fault, then the control flow goes to the fault handler, and no single-step debug exception is asserted. In such a situation, if the instruction is executed in VMX non-root operation and the “monitor trap flag” VM-execution control is 1, an MTF VM exit is pending after the delivery of the fault (or any nested exception). No MTF VM exit occurs if another VM exit occurs before reaching that boundary on which an MTF VM exit would be pending.

### 42.2.3 Single-Stepping Enclave Entry with Opt-out Entry

#### 42.2.3.1 Single Stepping without AEX

Figure 42-1 shows the most common case for single-stepping after an opt-out entry.

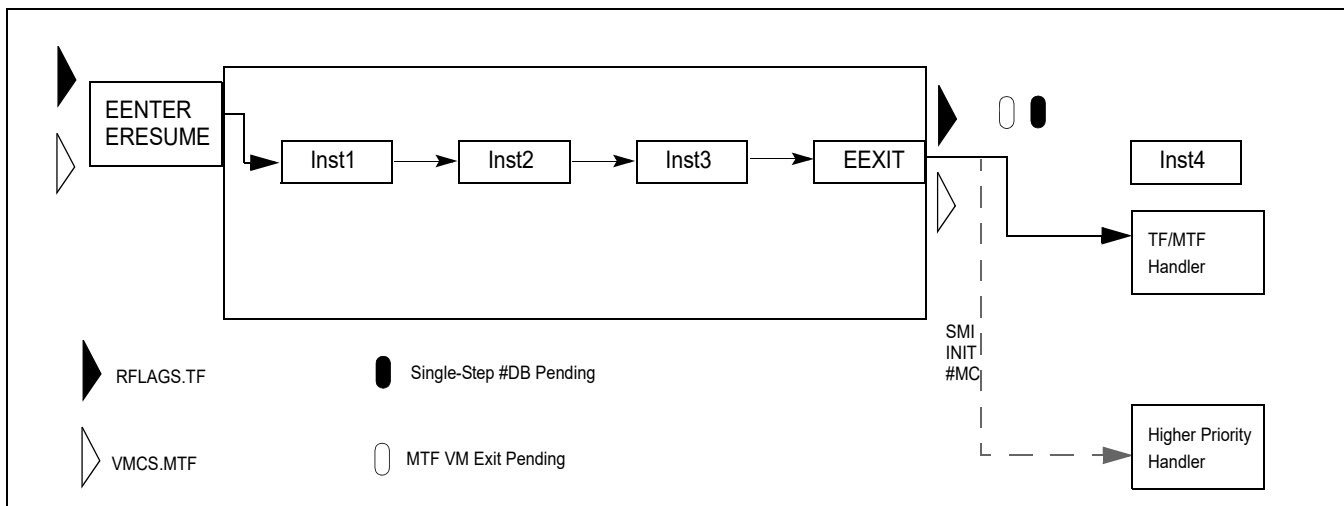


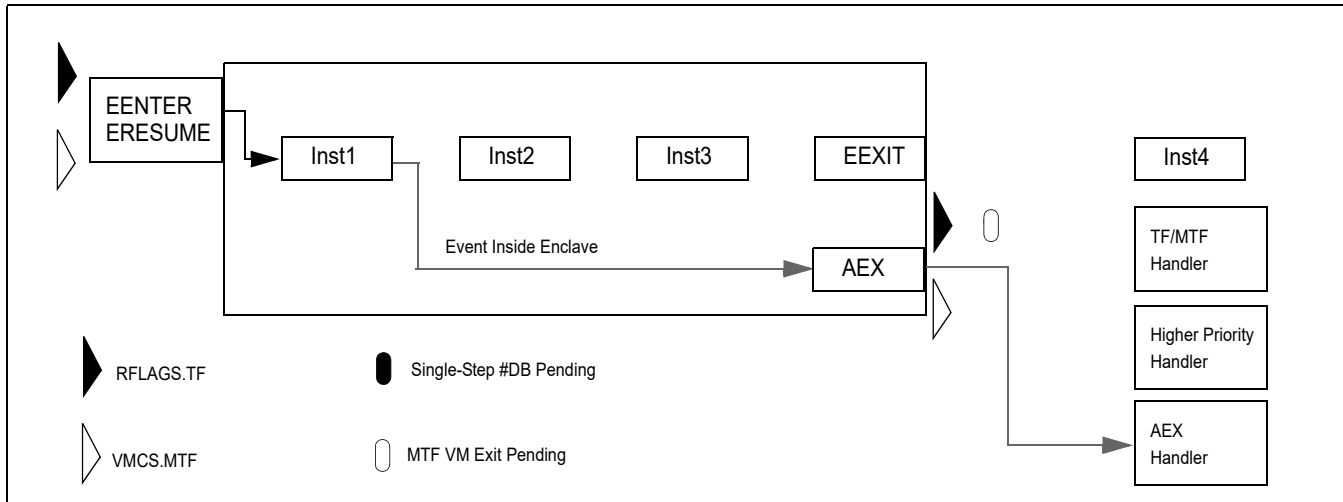
Figure 42-1. Single Stepping with Opt-out Entry - No AEX

In this scenario, if the RFLAGS.TF bit is set at the time of the enclave entry, then a single step debug exception is pending on the instruction boundary after EEXIT. Additionally, if the enclave is executing in VMX non-root operation and the “monitor trap flag” VM-execution control is 1, an MTF VM exit is pending on the instruction boundary after EEXIT.

The value of the RFLAGS.TF bit at the end of EEXIT is the same as the value of RFLAGS.TF at the time of the enclave entry.

#### 42.2.3.2 Single Step Preempted by AEX Due to Non-SMI Event

Figure 42-2 shows the interaction of single stepping with AEX due to a non-SMI event after an opt-out entry.



**Figure 42-2. Single Stepping with Opt-out Entry -AEX Due to Non-SMI Event Before Single-Step Boundary**

In this scenario, if the enclave is executing in VMX non-root operation and the “monitor trap flag” VM-execution control is 1, an MTF VM exit is pending on the instruction boundary after the AEX. No MTF VM exit occurs if another VM exit happens before reaching that instruction boundary.

The value of the RFLAGS.TF bit at the end of AEX is the same as the value of RFLAGS.TF at the time of the enclave entry.

#### 42.2.4 RFLAGS.TF Treatment on AEX

The value of EFLAGS.TF at the end of AEX from an opt-out enclave is same as the value of EFLAGS.TF at the time of the enclave entry. The value of EFLAGS.TF at the end of AEX from an opt-in enclave is unmodified. The EFLAGS.TF saved in GPR portion of the SSA on an AEX is 0. For more detail see EENTER and ERESUME in Chapter 5.

#### 42.2.5 Restriction on Setting of TF after an Opt-Out Entry

Enclave entered through an opt-out entry is not allowed to set EFLAGS.TF. The POPF instruction forces RFLAGS.TF to 0 if the enclave was entered through opt-out entry.

#### 42.2.6 Trampoline Code Considerations

Any AEX from the enclave which results in the RFLAGS.TF = 1 on the reporting stack will result in a single-step #DB after the first instruction of the trampoline code if the trampoline is entered using the IRET instruction.

### 42.3 CODE AND DATA BREAKPOINTS

#### 42.3.1 Breakpoint Suppression

Following an opt-out entry:

- Instruction breakpoints are suppressed during execution in an enclave.
- Data breakpoints are not triggered on accesses to the address range defined by ELRANGE.
- Data breakpoints are triggered on accesses to addresses outside the ELRANGE



Following an opt-in entry instruction and data breakpoints are not suppressed.

The processor does not report any matches on debug breakpoints that are suppressed on enclave entry. However, the processor does not clear any bits in DR6 that were already set at the time of the enclave entry.

### 42.3.2 Reporting of Instruction Breakpoint on Next Instruction on a Debug Trap

A debug exception caused by the single-step execution mode or when a data breakpoint condition was met causes the processor to perform an AEX. Following such an AEX, the processor reports in the debug status register (DR6) matches of the new instruction pointer (the AEP address) in a breakpoint address register setup to detect instruction execution.

### 42.3.3 RF Treatment on AEX

RF flag value saved in SSA is the same as what would have been pushed on stack if the exception or event causing the AEX occurred when executing outside an enclave (see Section 17.3.1.1). Following an AEX, the RF flag is 0 in the synthetic state.

### 42.3.4 Breakpoint Matching in Intel® SGX Instruction Flows

Implicit accesses made by Intel SGX instructions to EPC regions do not trigger data breakpoints. Explicit accesses made by ENCLS[ECREATE], ENCLS[EADD], ENCLS[EEXTEND], ENCLS[EINIT], ENCLS[EREMOVE], ENCLS[ETRACK], ENCLS[EBLOCK], ENCLS[EPA], ENCLS[EWB], ENCLS[ELD], ENCLS[EDBGGRD], ENCLS[EDBGWR], ENCLU[EENTER], and ENCLU[ERESUME] to the EPC operands do not trigger data breakpoints.

Explicit accesses made by the Intel SGX instructions (ENCLU[EGETKEY] and ENCLU[EREPORT]) executed by an enclave following an opt-in entry, trigger data breakpoints on accesses to their EPC operands. All Intel SGX instructions trigger data breakpoints on accesses to their non-EPC operands.

## 42.4 CONSIDERATION OF THE INT1 AND INT3 INSTRUCTIONS

This section considers the operation of the INT1 and INT3 instructions when executed inside an enclave. These are the instructions with opcodes F1 and CC, respectively, and not INT *n* (with opcode CD) with value 1 or 3 for *n*.

### 42.4.1 Behavior of INT1 and INT3 Inside an Enclave

An execution of either INT1 or INT3 inside an enclave results in a fault-class exception. Following an opt-out entry, execution of either instruction results in an invalid-opcode exception (#UD). Following opt-in entry, INT1 results in a debug exception (#DB) and INT3 delivers a breakpoint exception (#BP). The normal requirement for INT3 (that the CPL not be greater than the DPL of descriptor 3 in the IDT) is not enforced.

Because execution of INT1 or INT3 inside an enclave results in a fault, the RIP saved in the SSA on AEX references the INT1 or INT3 instruction (and not the following instruction). The RIP value saved on the stack (or in the TSS or VMCS) is that of the AEP.

If execution of INT1 or INT3 inside an enclave causes a VM exit, the event type in the VM-exit interruption information field indicates a hardware exception (type 3),<sup>1</sup> and the VM-exit instruction length field is saved as zero.

### 42.4.2 Debugger Considerations

A debugger using INT3 inside an enclave should account for the modified behavior described in Section 42.4.1. Because INT3 is fault-like inside an enclave, the RIP saved in the SSA on AEX is that of the INT3 instruction. Conse-

1. INT1 would normally indicate a privileged software exception (type 5), and INT3 would normally indicate a software exception (type 6).

quently, the debugger must not decrement SSA.RIP for #BP coming from an enclave to re-execute the instruction at the RIP of the INT3 instruction on a subsequent enclave entry.

### 42.4.3 VMM Considerations

As described in Section 42.4.1, execution of INT3 inside an enclave delivers #BP with “interruption type” of 3. A VMM that re-injects #BP into the guest should establish the VM-entry interruption information field using data saved into the appropriate VMCS fields by the VM exit incident to the #BP (as recommended in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*).

VMMs that create the VM-entry interruption information based solely on the exception vector should take care to use event type 3 (instead of 6) when they detect a VM exit incident to enclave mode that is due to an exception with vector 3.

## 42.5 BRANCH TRACING

### 42.5.1 BTF Treatment

When software enables single-stepping on branches then:

- Following an opt-in entry using EENTER the processor generates a single step debug exception.
- Following an EEXIT the processor generates a single-step debug exception

Enclave entry using ERESUME (opt-in or opt-out) and an AEX from the enclave do not cause generation of the single-step debug exception.

### 42.5.2 LBR Treatment

#### 42.5.2.1 LBR Stack on Opt-in Entry

Following an opt-in entry into an enclave, last branch recording facilities if enabled continued to store branch records in the LBR stack MSRs as follows:

- On enclave entry using EENTER/ERESUME, the processor push the address of EENTER/ERESUME instruction into MSR\_LASTBRANCH\_n\_FROM\_IP, and the destination address of the EENTER/ERESUME into MSR\_LASTBRANCH\_n\_TO\_IP.
- On EEXIT, the processor pushes the address of EEXIT instruction into MSR\_LASTBRANCH\_n\_FROM\_IP, and the address of EEXIT destination into MSR\_LASTBRANCH\_n\_TO\_IP.
- On AEX, the processor pushes RIP saved in the SSA into MSR\_LASTBRANCH\_n\_FROM\_IP, and the address of AEP into MSR\_LASTBRANCH\_n\_TO\_IP.
- For every branch inside the enclave, a branch record is pushed on the LBR stack.

Figure 42-3 shows an example of LBR stack manipulation after an opt-in entry. Every arrow in this picture indicates a branch record pushed on the LBR stack. The “From IP” of the branch record contains the linear address of the instruction located at the start of the arrow, while the “To IP” of the branch record contains the linear address of the instruction at the end of the arrow.

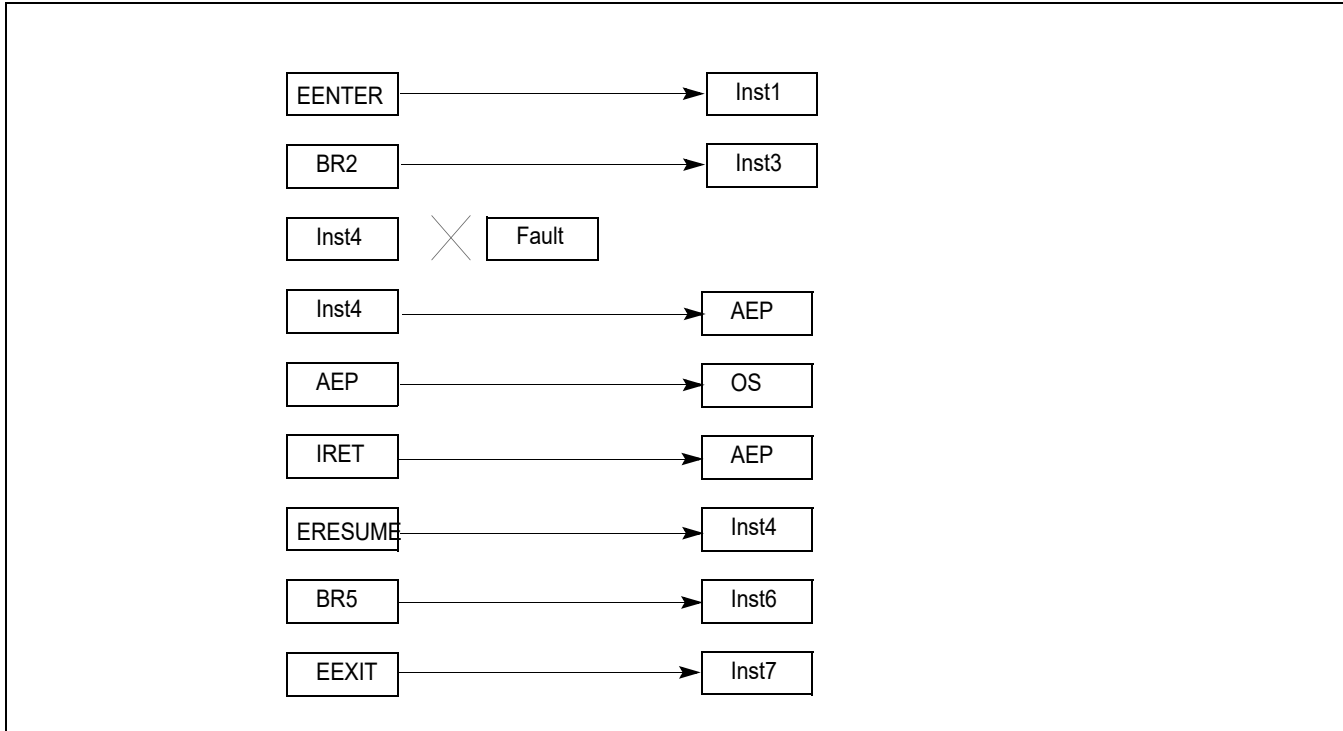


Figure 42-3. LBR Stack Interaction with Opt-in Entry

### 42.5.2.2 LBR Stack on Opt-out Entry

An opt-out entry into an enclave suppresses last branch recording facilities, and enclave exit after an opt-out entry un-suppresses last branch recording facilities.

Opt-out entry into an enclave does not push any record on LBR stack.

If last branch recording facilities were enabled at the time of enclave entry, then EEXIT following such an enclave entry pushes one record on LBR stack. The MSR\_LASTBRANCH\_n\_FROM\_IP of such record holds the linear address of the instruction (EENTER or ERESUME) that was used to enter the enclave, while the MSR\_LASTBRANCH\_n\_TO\_IP of such record holds linear address of the destination of EEXIT.

Additionally, if last branch recording facilities were enabled at the time of enclave entry, then an AEX after such an entry pushes one record on LBR stack, before pushing record for the event causing the AEX if the event pushes a record on LBR stack. The MSR\_LASTBRANCH\_n\_FROM\_IP of the new record holds linear address of the instruction (EENTER or ERESUME) that was used to enter the enclave, while MSR\_LASTBRANCH\_n\_TO\_IP of the new record holds linear address of the AEP. If the event causing AEX pushes a record on LBR stack, then the MSR\_LASTBRANCH\_n\_FROM\_IP for that record holds linear address of the AEP.

Figure 42-4 shows an example of LBR stack manipulation after an opt-out entry. Every arrow in this picture indicates a branch record pushed on the LBR stack. The “From IP” of the branch record contains the linear address of the instruction located at the start of the arrow, while the “To IP” of the branch record contains the linear address of the instruction at the end of the arrow.

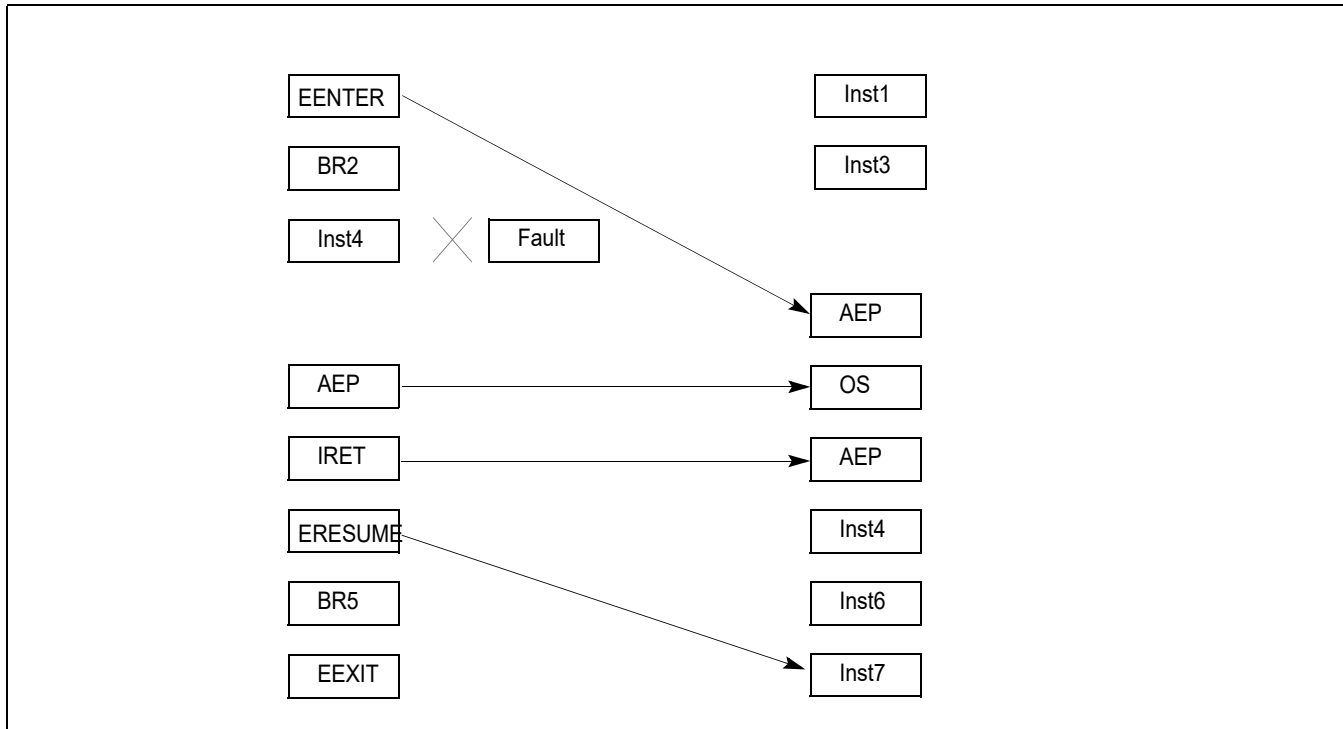


Figure 42-4. LBR Stack Interaction with Opt-out Entry

### 42.5.2.3 Mispredict Bit, Record Type, and Filtering

All branch records resulting from Intel SGX instructions/AEXs are reported as predicted branches, and consequently, bit 63 of MSR\_LASTBRANCH\_n\_FROM\_IP for such records is set. Branch records due to these Intel SGX operations are always non-HLE/non-RTM records.

EENTER, ERESUME, EEXIT, and AEX are considered to be far branches. Consequently, bit 8 in MSR\_LBR\_SELECT controls filtering of the new records introduced by Intel SGX.

## 42.6 INTERACTION WITH PERFORMANCE MONITORING

### 42.6.1 IA32\_PERF\_GLOBAL\_STATUS Enhancement

On processors supporting Intel SGX, the IA32\_PERF\_GLOBAL\_STATUS MSR provides a bit indicator, known as “Anti Side-channel Interference” (ASCI) at bit position 60. If this bit is 0, the performance monitoring data in various performance monitoring counters are accumulated normally as defined by relevant architectural/microarchitectural conditions. If the ASCI bit is set, the contents in various performance monitoring counters can be affected by the direct or indirect consequence of Intel SGX protection of enclave code executing in the processor.

### 42.6.2 Performance Monitoring with Opt-in Entry

An opt-in enclave entry allow performance monitoring logic to observe the contribution of enclave code executing in the processor. Thus the contents of performance monitoring counters does not distinguish between contribution originating from enclave code or otherwise. All counters, events, precise events, etc. continue to work as defined in the IA32/Intel 64 Software Developer Manual. Consequently, bit 60 of IA32\_PERF\_GLOBAL\_STATUS MSR is not set.

### 42.6.3 Performance Monitoring with Opt-out Entry

In general, performance monitoring activities are suppressed when entering an opt-out enclave. This applies to all thread-specific, configured performance monitoring, except for the cycle-counting fixed counter, IA32\_FIXED\_CTR1 and IA32\_FIXED\_CTR2. Upon entering an opt-out enclave, IA32\_FIXED\_CTR0, IA32\_PMCx will stop accumulating counts. Additionally, if PEBS is configured to capture PEBS record for this thread, PEBS record generation will also be suppressed. Consequently, bit 60 of IA32\_PERF\_GLOBAL\_STATUS MSR is set.

Performance monitoring on the sibling thread may also be affected. Any one of IA32\_FIXED\_CTRx or IA32\_PMCx on the sibling thread configured to monitor thread-specific eventing logic with AnyThread = 1 is demoted to count only MyThread while an opt-out enclave is executing on the other thread.

### 42.6.4 Enclave Exit and Performance Monitoring

When a logical processor exits an enclave, either via ENCLU[EEXIT] or via AEX, all performance monitoring activity (including PEBS) on that logical processor that was suppressed is unsuppressed.

Any counters that were demoted from AnyThread to MyThread on the sibling thread are promoted back to AnyThread.

### 42.6.5 PEBS Record Generation on Intel® SGX Instructions

All leaf functions of the ENCLS instruction report "Eventing RIP" of the ENCLS instruction if a PEBS record is generated at the end of the instruction execution. Additionally, the EGETKEY and EREPORT leaf functions of the ENCLU instruction report "Eventing RIP" of the ENCLU instruction if a PEBS record is generated at the end of the instruction execution.

If the EENTER and ERESUME leaf functions are performing an opt-in entry report "Eventing RIP" of the ENCLU instruction if a PEBS record is generated at the end of the instruction execution. On the other hand, if these leaf functions are performing an opt-out entry, then these leaf functions result in PEBS being suppressed, and no PEBS record is generated at the end of these instructions.

A PEBS record is generated if there is a PEBS event pending at the end of EEXIT (due to a counter overflowing during enclave execution or during EEXIT execution). This PEBS record contains the architectural state of the logical processor at the end of EEXIT. If the enclave was entered via an opt-in entry, then this record reports the "Eventing RIP" as the linear address of the ENCLU[EEXIT] instruction. If the enclave was entered via an opt-out entry, then the record reports the "Eventing RIP" as the linear address of the ENCLU[EENTER/ERESUME] instruction that performed the last enclave entry.

A PEBS record is generated after the AEX if there is a PEBS event pending at the end of AEX (due to a counter overflowing during enclave execution or during AEX execution). This PEBS record contains the synthetic state of the logical processor that is established at the end of AEX. For opt-in entry, this record has the EVENTING\_RIP set to the RIP saved in the SSA. For opt-out entry, the record has the EVENTING\_RIP set to the linear address of EENTER/ERESUME used for the last enclave entry.

If the enclave was entered via an opt-in entry, then this record reports the "Eventing RIP" as the linear address in the SSA of the enclave (a.k.a., the "Eventing LIP" inside the enclave). If the enclave was entered via an opt-out entry, then the record reports the "Eventing RIP" as the linear address of the ENCLU[EENTER/ERESUME] instruction that performed the last enclave entry.

A second PEBS event may be pended during the Enclave Exiting Event (EEE). If the PEBS event is taken at the end of delivery of the EEE then the "Eventing RIP" in this second PEBS record is the linear address of the AEP.

### 42.6.6 Exception-Handling on PEBS/BTS Loads/Stores after AEX

The operating system should allocate sections of the DS save area from a non-paged pool, and mark them as accessed and dirty. If the loads/stores to any section of the DS save area incur faults then such faults are reported to the OS/VMM immediately, and generation of the PEBS/BTS record is skipped and may leave the buffers in a state where they have a partial PEBS or BTS records.

However, any events that are detected during PEBS/BTS record generation at the end of AEX and before delivering the Enclave Exiting Event (EEE) cannot be reported immediately to the OS/VMM, as an event window is not open at

the end of AEX. Consequently, fault-like events such as page faults, EPT faults, EPT mis-configuration, and accesses to APIC-access page detected on stores to the PEBS/BTS buffer are not reported, and generation of the PEBS and/or BTS record at the end of AEX is aborted (this may leave the buffers in a state where they have partial PEBS or BTS records). Trap-like events detected on stores to the PEBS/BTS buffer (such as debug traps) are pended until the next instruction boundary, where they are handled according to the architecturally defined priority. The processor continues the handling of the Enclave Exiting Event (SMI, NMI, interrupt, exception delivery, VM exit, etc.) after aborting the PEBS/BTS record generation.

#### **42.6.6.1 Other Interactions with Performance Monitoring**

For opt-in entry, EENTER, ERESUME, EEXIT, and AEX are all treated as predicted far branches, and any counters that are counting such branches are incremented by 1 as a part of retirement of these instructions. Retirement of these instructions is also counted in any counters configured to count instructions retired.

For opt-out entry, execution inside an enclave is treated as a single predicted branch, and all branch-counting performance monitoring counters are incremented accordingly. Additionally, such execution is also counted as a single instruction, and all performance monitoring counters counting instructions are incremented accordingly.

Enclave entry does not affect any performance monitoring counters shared between cores.



## 24. Updates to Appendix C, Volume 3D

Change bars show changes to Appendix C of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*.

-----  
Changes to this chapter: Update to Table C-1 "Basic Exit Reasons".



## APPENDIX C VMX BASIC EXIT REASONS

Every VM exit writes a 32-bit exit reason to the VMCS (see Section 24.9.1). Certain VM-entry failures also do this (see Section 26.7). The low 16 bits of the exit-reason field form the basic exit reason which provides basic information about the cause of the VM exit or VM-entry failure.

Table C-1 lists values for basic exit reasons and explains their meaning. Entries apply to VM exits, unless otherwise noted.

**Table C-1. Basic Exit Reasons**

Basic Exit Reason	Description
0	<b>Exception or non-maskable interrupt (NMI).</b> Either: 1: Guest software caused an exception and the bit in the exception bitmap associated with exception's vector was 1. This case includes executions of BOUND that cause #BR, executions of INT1 (they cause #DB), executions of INT3 (they cause #BP), executions of INTO that cause #OF, and executions of UDO, UD1, and UD2 (they cause #UD). 2: An NMI was delivered to the logical processor and the "NMI exiting" VM-execution control was 1.
1	<b>External interrupt.</b> An external interrupt arrived and the "external-interrupt exiting" VM-execution control was 1.
2	<b>Triple fault.</b> The logical processor encountered an exception while attempting to call the double-fault handler and that exception did not itself cause a VM exit due to the exception bitmap.
3	<b>INIT signal.</b> An INIT signal arrived
4	<b>Start-up IPI (SIPI).</b> A SIPI arrived while the logical processor was in the "wait-for-SIPI" state.
5	<b>I/O system-management interrupt (SMI).</b> An SMI arrived immediately after retirement of an I/O instruction and caused an SMM VM exit (see Section 34.15.2).
6	<b>Other SMI.</b> An SMI arrived and caused an SMM VM exit (see Section 34.15.2) but not immediately after retirement of an I/O instruction.
7	<b>Interrupt window.</b> At the beginning of an instruction, RFLAGS.IF was 1; events were not blocked by STI or by MOV SS; and the "interrupt-window exiting" VM-execution control was 1.
8	<b>NMI window.</b> At the beginning of an instruction, there was no virtual-NMI blocking; events were not blocked by MOV SS; and the "NMI-window exiting" VM-execution control was 1.
9	<b>Task switch.</b> Guest software attempted a task switch.
10	<b>CPUID.</b> Guest software attempted to execute CPUID.
11	<b>GETSEC.</b> Guest software attempted to execute GETSEC.
12	<b>HLT.</b> Guest software attempted to execute HLT and the "HLT exiting" VM-execution control was 1.
13	<b>INVD.</b> Guest software attempted to execute INVD.
14	<b>INVLPG.</b> Guest software attempted to execute INVLPG and the "INVLPG exiting" VM-execution control was 1.
15	<b>RDPMC.</b> Guest software attempted to execute RDPMC and the "RDPMC exiting" VM-execution control was 1.
16	<b>RDTSC.</b> Guest software attempted to execute RDTSC and the "RDTSC exiting" VM-execution control was 1.
17	<b>RSM.</b> Guest software attempted to execute RSM in SMM.
18	<b>VMCALL.</b> VMCALL was executed either by guest software (causing an ordinary VM exit) or by the executive monitor (causing an SMM VM exit; see Section 34.15.2).
19	<b>VMCLEAR.</b> Guest software attempted to execute VMCLEAR.
20	<b>VMLAUNCH.</b> Guest software attempted to execute VMLAUNCH.
21	<b>VMPTRLD.</b> Guest software attempted to execute VMPTRLD.
22	<b>VMPTRST.</b> Guest software attempted to execute VMPTRST.

**Table C-1. Basic Exit Reasons (Contd.)**

Basic Exit Reason	Description
23	<b>VMREAD.</b> Guest software attempted to execute VMREAD.
24	<b>VMRESUME.</b> Guest software attempted to execute VMRESUME.
25	<b>VMWRITE.</b> Guest software attempted to execute VMWRITE.
26	<b>VMXOFF.</b> Guest software attempted to execute VMXOFF.
27	<b>VMXON.</b> Guest software attempted to execute VMXON.
28	<b>Control-register accesses.</b> Guest software attempted to access CR0, CR3, CR4, or CR8 using CLTS, LMSW, or MOV CR and the VM-execution control fields indicate that a VM exit should occur (see Section 25.1 for details). This basic exit reason is not used for trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1. Such VM exits instead use basic exit reason 43.
29	<b>MOV DR.</b> Guest software attempted a MOV to or from a debug register and the “MOV-DR exiting” VM-execution control was 1.
30	<b>I/O instruction.</b> Guest software attempted to execute an I/O instruction and either: 1: The “use I/O bitmaps” VM-execution control was 0 and the “unconditional I/O exiting” VM-execution control was 1. 2: The “use I/O bitmaps” VM-execution control was 1 and a bit in the I/O bitmap associated with one of the ports accessed by the I/O instruction was 1.
31	<b>RDMSR.</b> Guest software attempted to execute RDMSR and either: 1: The “use MSR bitmaps” VM-execution control was 0. 2: The value of RCX is neither in the range 00000000H - 00001FFFH nor in the range C0000000H - C0001FFFH. 3: The value of RCX was in the range 00000000H - 00001FFFH and the $n^{\text{th}}$ bit in read bitmap for low MSRs is 1, where $n$ was the value of RCX. 4: The value of RCX is in the range C0000000H - C0001FFFH and the $n^{\text{th}}$ bit in read bitmap for high MSRs is 1, where $n$ is the value of RCX & 00001FFFH.
32	<b>WRMSR.</b> Guest software attempted to execute WRMSR and either: 1: The “use MSR bitmaps” VM-execution control was 0. 2: The value of RCX is neither in the range 00000000H - 00001FFFH nor in the range C0000000H - C0001FFFH. 3: The value of RCX was in the range 00000000H - 00001FFFH and the $n^{\text{th}}$ bit in write bitmap for low MSRs is 1, where $n$ was the value of RCX. 4: The value of RCX is in the range C0000000H - C0001FFFH and the $n^{\text{th}}$ bit in write bitmap for high MSRs is 1, where $n$ is the value of RCX & 00001FFFH.
33	<b>VM-entry failure due to invalid guest state.</b> A VM entry failed one of the checks identified in Section 26.3.1.
34	<b>VM-entry failure due to MSR loading.</b> A VM entry failed in an attempt to load MSRs. See Section 26.4.
36	<b>MWAIT.</b> Guest software attempted to execute MWAIT and the “MWAIT exiting” VM-execution control was 1.
37	<b>Monitor trap flag.</b> A VM entry occurred due to the 1-setting of the “monitor trap flag” VM-execution control and injection of an MTF VM exit as part of VM entry. See Section 25.5.2.
39	<b>MONITOR.</b> Guest software attempted to execute MONITOR and the “MONITOR exiting” VM-execution control was 1.
40	<b>PAUSE.</b> Either guest software attempted to execute PAUSE and the “PAUSE exiting” VM-execution control was 1 or the “PAUSE-loop exiting” VM-execution control was 1 and guest software executed a PAUSE loop with execution time exceeding PLE_Window (see Section 25.1.3).
41	<b>VM-entry failure due to machine-check event.</b> A machine-check event occurred during VM entry (see Section 26.8).
43	<b>TPR below threshold.</b> The logical processor determined that the value of bits 7:4 of the byte at offset 080H on the virtual-APIC page was below that of the TPR threshold VM-execution control field while the “use TPR shadow” VM-execution control was 1 either as part of TPR virtualization (Section 29.1.2) or VM entry (Section 26.6.7).
44	<b>APIC access.</b> Guest software attempted to access memory at a physical address on the APIC-access page and the “virtualize APIC accesses” VM-execution control was 1 (see Section 29.4).
45	<b>Virtualized EOI.</b> EOI virtualization was performed for a virtual interrupt whose vector indexed a bit set in the EOI-exit bitmap.

Table C-1. Basic Exit Reasons (Contd.)

Basic Exit Reason	Description
46	<b>Access to GDTR or IDTR.</b> Guest software attempted to execute LGDT, LIDT, SGDT, or SIDT and the “descriptor-table exiting” VM-execution control was 1.
47	<b>Access to LDTR or TR.</b> Guest software attempted to execute LLDT, LTR, SLDT, or STR and the “descriptor-table exiting” VM-execution control was 1.
48	<b>EPT violation.</b> An attempt to access memory with a guest-physical address was disallowed by the configuration of the EPT paging structures.
49	<b>EPT misconfiguration.</b> An attempt to access memory with a guest-physical address encountered a misconfigured EPT paging-structure entry.
50	<b>INVEPT.</b> Guest software attempted to execute INVEPT.
51	<b>RDTSCP.</b> Guest software attempted to execute RDTSCP and the “enable RDTSCP” and “RDTSC exiting” VM-execution controls were both 1.
52	<b>VMX-preemption timer expired.</b> The preemption timer counted down to zero.
53	<b>INVPID.</b> Guest software attempted to execute INVVPID.
54	<b>WBINVD.</b> Guest software attempted to execute WBINVD and the “WBINVD exiting” VM-execution control was 1.
55	<b>XSETBV.</b> Guest software attempted to execute XSETBV.
56	<b>APIC write.</b> Guest software completed a write to the virtual-APIC page that must be virtualized by VMM software (see Section 29.4.3.3).
57	<b>RDRAND.</b> Guest software attempted to execute RDRAND and the “RDRAND exiting” VM-execution control was 1.
58	<b>INVPCID.</b> Guest software attempted to execute INVPCID and the “enable INVPCID” and “INVLPG exiting” VM-execution controls were both 1.
59	<b>VMFUNC.</b> Guest software invoked a VM function with the VMFUNC instruction and the VM function either was not enabled or generated a function-specific condition causing a VM exit.
60	<b>ENCLS.</b> Guest software attempted to execute ENCLS and “enable ENCLS exiting” VM-execution control was 1 and either (1) EAX < 63 and the corresponding bit in the ENCLS-exiting bitmap is 1; or (2) EAX ≥ 63 and bit 63 in the ENCLS-exiting bitmap is 1.
61	<b>RDSEED.</b> Guest software attempted to execute RDSEED and the “RDSEED exiting” VM-execution control was 1.
62	<b>Page-modification log full.</b> The processor attempted to create a page-modification log entry and the value of the PML index was not in the range 0-511.
63	<b>XSAVES.</b> Guest software attempted to execute XSAVES, the “enable XSAVES/XRSTORS” was 1, and a bit was set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
64	<b>XRSTORS.</b> Guest software attempted to execute XRSTORS, the “enable XSAVES/XRSTORS” was 1, and a bit was set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.

